Practical Refinement-Type Checking

Rowan Davies

CMU-CS-05-110
May, 2005

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee

Frank Pfenning, Chair
Robert Harper
Peter Lee
John Reynolds
Alex Aiken, Stanford University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was sponsored in part by the National Science Foundation under grant no. CCR-0204248, and in part by a Hackett Studentship from the University of Western Australia. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Programming languages, type systems, intersection types, bidirectional checking, value restricted polymorphism, regular-tree types.

Abstract

Software development is a complex and error prone task. Programming languages with strong static type systems assist programmers by capturing and checking the fundamental structure of programs in a very intuitive way. Given this success, it is natural to ask: can we capture and check more of the structure of programs?

In this dissertation I describe an approach called *refinement-type checking* that allows many common program properties to be captured and checked. This approach builds on the strength of the type system of a language by adding the ability to specify refinements of each type. Following previous work, I focus on refinements that include subtyping and a form of intersection types.

Central to my approach is the use of a bidirectional checking algorithm. This does not attempt to infer refinements for some expressions, such as functions, but only checks them against refinements. This avoids some difficulties encountered in previous work, and requires that the programmer annotate their program with some of the intended refinements. The required annotations appear to be very reasonable. Further, they document properties in a way that is natural, precise, easy to read, and reliable.

I demonstrate the practicality of my approach by showing that it can be used to design a refinement-type checker for a widely-used language with a strong type system: Standard ML. This requires two main technical developments. Firstly, I present a new variant of intersection types that obtain soundness in the presence of call-by-value effects by incorporating a value restriction. Secondly, I present a practical approach to incorporating recursive refinements of ML datatypes, including a pragmatic method for checking the sequential pattern matching construct of ML.

I conclude by reporting the results of experiments with my implementation of refinement-type checking for SML. These indicate that refinement-type checking is a practical method for capturing and checking properties of real code.

Contents

Acknowledgments

Firstly, I would like to thank Frank for his advice, encouragement, patience and wisdom. Many of the ideas upon which this work is built are his, including the original concept of refinement types and the idea of checking them bidirectionally, and he was involved at every step in the process. I have certainly learnt far more than I expected during my journey through the Ph.D. program in Pure and Applied Logic, and I feel very privileged to have had the opportunity to learn from a true master of the field.

I would also like to thank my committee members for their patience as well as their input and feedback over the years which have certainly improved the final product. Bob Harper deserves particular thanks for his insightful input on a broad range of issues, both in the early stages of this work, and in detailed feedback closer to the end.

Next, my special thanks to Sharon Burks for both her understanding and also her assistance with the various arrangements required in order for me to complete this work.

Finally, I would like to thank my family and friends for their support, their assistance in putting things in perspective, and, at the risk of sounding repetitive, their patience as well.

Chapter 1

Introduction

1.1 Thesis

A new technique called refinement-type checking provides a practical mechanism for expressing and verifying many properties of programs written in fully featured languages.

1.2 Motivation

Static type systems are a central feature of many programming languages. They provide a natural and intuitive mechanism for expressing and checking the fundamental structure of programs. They thus allow many errors in programs to be automatically detected at an early stage, and they significantly aid the process of understanding unfamiliar code. This is particularly true for large, modular programs, since types can be used to describe module interfaces.

While strong static type systems are very effective at capturing the basic structure of a program, generally programs involve many important properties that are not captured by types. For example, a particular function may always return a non-zero number, or may require that its argument be non-zero, but programming languages generally do not provide a specific type for non-zero numbers.

Such invariants are often critical to the understanding and correctness of a program, but usually they are only informally documented via comments. While such comments are certainly useful, it takes considerable discipline to ensure that properties are described accurately and precisely, particularly when the code may be modified frequently. Further, the lack of any convenient mechanism for checking whether the code actually satisfies the stated properties means that such comments cannot be relied upon.

We might consider attempting to construct formal proofs that such properties are satisfied. However, constructing such proofs is generally difficult or infeasible. Further, as a program evolves, the proof needs to be evolved, which is likely to be awkward. Additionally, when the intended properties do not hold due to an error in the code, it is unlikely that this method will guide the programmer to the source of the error as quickly as the error messages produced by a type checker.

This work demonstrates that a new approach to capturing and checking some of these properties can be used to build practical tools. This approach builds on the strength of the

static type system of a language by adding the ability to specify *refinements* of each type. These *refinement types* include constructs which follow the structure of the type that they refine, and additionally include features that are particularly appropriate for specifying program properties.

1.3 Background: refinement types

Refinement types were introduced by Freeman and Pfenning [FP91]. They do not require altering the type system of a language: instead we add a new kind of checking which follows the structure of the type system, but additionally includes features that are appropriate for expressing and checking properties of programs. This means that we are conservatively extending the language: all programs in the original language are accepted as valid programs in our extension.

We also refer to refinement types as *sorts* in accordance with the use of this term in order-sorted algebras [DG94]. This allows us to use convenient terminology that mirrors that for types: we can use terms such as subsorting and sort checking and thus make a clear link with the corresponding notions for types.

We illustrate the features of sorts with a running example. We first illustrate the constructs that follow the structure of types, focusing on the situation for functions. Suppose we have a sort pos for positive integers which refines a type num for numbers. Then, we can form a sort for functions mapping positive integers to positive integers: $\mathsf{pos} \to \mathsf{pos}$. This uses the construct $\to$ which mirrors the corresponding construct for types. If we have an additional refinement nat for natural numbers, then we can form the following refinements of the type $\mathsf{num} \to \mathsf{num}$.

$$\mathsf{pos} \to \mathsf{pos} \qquad \mathsf{pos} \to \mathsf{nat} \qquad \mathsf{nat} \to \mathsf{pos} \qquad \mathsf{nat} \to \mathsf{nat}$$

Sorts include similar constructs mirroring each type construct. We now consider the other features of sorts, which are included specifically because they are appropriate for capturing program properties.

Sorts express properties of programs, and generally there are natural inclusion relationships between these properties. For example, every positive number is a natural number, so we should allow a positive number whenever a natural number is required. Thus, we have a natural partial order on the refinements of each type, and we write $\mathsf{pos} \leq \mathsf{nat}$ to indicate this order. This is essentially a form of subtyping, although we refer to it as *subsorting* since the order is on the refinements of a particular type rather than on the types themselves. This partial order is extended to refinements of function types following the standard contravariant-covariant subtyping rule. Thus, the following inclusion holds.

$$\mathsf{nat} \to \mathsf{pos} \leq \mathsf{pos} \to \mathsf{nat}$$

In practice it is sometimes necessary to assign more than one property to a particular part of a program. For example, if we have a function double with type $\mathsf{num} \to \mathsf{num}$ that doubles a number, we may need two properties of this function: that it maps positive numbers to positive numbers, and that it maps natural numbers to natural numbers. To allow multiple properties to be specified in such situations, sorts include an *intersection* operator & which allows two refinements of the same type to be combined. Thus, we can specify the desired property of

double with the following sort.

$$(\mathsf{pos} \to \mathsf{pos}) \;\&\; (\mathsf{nat} \to \mathsf{nat})$$

The operator & is based on work on intersection types, such as that of Coppo, Dezani-Ciancaglini and Venneri [CDV81] and Reynolds [Rey96].

One might notice that refinements are essentially another level of types, and wonder whether it is really necessary to have both ordinary types and refinements as two separate levels for the same language. In fact, it is possible to design a language which instead includes intersections and subtyping in the ordinary type system. We consider such a language in Chapter 4, and a real language with these features has been described by Reynolds [Rey96]. However, we have both philosophical and practical reasons for considering types and refinements as two separate levels.

The philosophical reason is that we consider type correctness to be necessary in order for the semantics of a program to be defined, while refinements only express properties of programs that have already been determined to be valid. This is essentially the distinction between typed languages in the style of Church [Chu40] and type assignment systems in the style of Curry [Cur34]. Reynolds [Rey02] has considered a similar distinction between intrinsic and extrinsic semantics. In our case we consider that we have both, with one system refining the other, and we would argue that this is a natural design since the two levels serve different purposes.

The practical reason for considering types and refinements as two separate levels is that it allows us to extend an existing widely-used typed language without modifying it in any way. This allows us to easily experiment with refinements in real code. It also allows others to use refinements without committing to writing code in an experimental language. Thus, this approach allows significant experience to be gained in programming with advanced features such as intersection types and subtyping without introducing a new language.

Sorts are of little use to a programmer without a practical tool for checking the sorts associated with a program. Previous work on sorts focused on algorithms for sort inference, but this seems to be problematic. One reason for this that code generally satisfies many accidental properties which must be reflected in the inferred sort. Such accidental properties often prevent errors from being reported appropriately, such as when a function is applied to an inappropriate argument that nevertheless matches part of the inferred sort for the function. Further, as we move to more complicated types there is a combinatorial explosion in the number of refinements and the potential size of principal sorts. Experiments with refinements have thus been limited to relatively small and simple code fragments in previous work.

1.4 Our approach

1.4.1 Bidirectional checking

Central to the work described in this dissertation is a new approach called *refinement-type checking*, which we also call *sort checking*. This approach uses a bidirectional algorithm that does not attempt to infer sorts for some forms of expressions, such as functions, but instead only checks them against sorts. We still infer sorts for other forms of expressions, such as variables

and applications. The technique is called bidirectional because it works top-down through a program when checking against sorts, and bottom-up when inferring sorts.

Bidirectional algorithms have been considered previously by Reynolds [Rey96] and Pierce [Pie97] for languages with general intersection types, and by Pierce and Turner [PT98] for a language with impredicative polymorphism and subtyping.

Generally bidirectional algorithms require that the programmer annotate some parts of their program with the intended sorts. In our case, these annotations are mostly only required for function definitions, and only those for which the programmer has in mind a property beyond what is checked by the ordinary type system. Experience so far suggests that this requirement is very reasonable in practice. Further, these annotations usually appear at locations where it is natural to describe the properties using a comment anyway. They thus document properties in a way that is natural, easy to read, and precise. Additionally, these annotations can be relied upon to a greater extent than informal comments, since they are mechanically verified by sort checking.

To demonstrate the practicality and utility of sort checking for real programs, we have designed and implemented a sort checker for Standard ML, which is a widely used programming language with a strong static type system and advanced support for modular programming [MTHM97]. We now briefly outline the two main technical developments required to extend our approach to this fully featured language. These form the technical core of this dissertation, along with our bidirectional approach to sort checking.

1.4.2 Intersection types with call-by-value effects

The first main technical development is a new form of intersection types that achieves soundness in the presence of call-by-value effects. The standard form of intersection types is unsound in the presence of such effects, as illustrated by the following SML code, which includes sort annotations in stylized comments (as used by our implementation).

```
(*[ cell <: (pos ref) & (nat ref) ]*)
val cell = ref one
val () = (cell := zero)

(*[ result <: pos ]*)
val result = !cell
```

Here we create a reference `cell` that initially contains `one`. (We assume that `one` and `zero` have the expected refinements `pos` and `nat`.) Since `one` has sort `pos` we can assign `ref one` the sort `pos ref`, and since `one` has sort `nat` we can assign `ref one` the sort `nat ref`. The standard rule for intersection introduction then allows us to assign `ref one` the intersection of these two sorts `(pos ref) & (nat ref)`.

This leads to unsoundness, because the first part of the intersection allows us to update `cell` with `zero`, while the second part of the intersection allows us to conclude that reading the contents of `cell` will only return values with sort `pos`. Hence, standard intersection types allow us to assign `result` the sort `pos` when it will actually be bound to the value `zero`, which is clearly incorrect.

Our solution to this problem is to restrict the introduction of intersections to values. Our restricted form of intersection introduction states that if we can assign a value V the sort R

and also the sort S then we can assign the intersection of those sorts $R\&S$. (The general form of the rule allows any expression, not just values.)

This follows the value restriction on parametric polymorphism in the revised definition of SML [MTHM97], which was first proposed by Wright and Felleisen [WF94]. In our case we find that we additionally need to remove one of the standard subtyping rules for intersection types. The resulting system has some pleasing properties, and seems even better suited to bidirectional checking than standard intersection types.

1.4.3 Datasorts and pattern matching

The second main technical development is a practical approach to refinements of ML datatypes, including the sort checking of sequential pattern matching. Following previous work on sorts, we focus on refinements which are introduced using a mechanism for refining datatypes using recursive definitions. These refinements are particularly appropriate for ML because datatypes play an important role in the language: e.g. conditional control-flow is generally achieved by pattern matching with datatypes.

We illustrate the expressiveness of these refinements with an example. Suppose we have a program that includes the following ML datatype for strings of bits.

```
datatype bits = bnil | b0 of bits | b1 of bits
```

Further, suppose that in part of the program this datatype is used to represent natural numbers with the least significant digits at the beginning of the string. To ensure that there is a unique representation of each number, the program uses the following representation invariant: a natural number should have no zeros in the most significant positions (i.e. at the end). We can capture this invariant with the following *datasort* declarations, which define refinements of the datatype `bits`.

```
(*[ datasort nat = bnil | b0 of pos | b1 of nat
          and pos =       b0 of pos | b1 of nat ]*)
```

The syntax for datasort declarations mirrors that that of the datatype declarations which are being refined, except that some value constructors may be omitted and some may appear more than once with different sorts for the constructor argument. The datasort `nat` represents valid natural numbers, while `pos` represents valid positive natural numbers. In this declaration, the datasort `pos` is necessary in order to define `nat`, since `b0 bnil` is not a valid representation of a natural number. The inclusion `pos` $\leq$ `nat` clearly holds for these two datasorts, just like the refinements of the type `num` that we considered in Section 1.3.

In general, we would like to determine which inclusions hold for a particular set of datasort declarations. We differ from previous work on refinements in that we formulate an algorithm for determining which inclusions hold that is complete with respect to an inductive semantics in the case when the recursive definitions correspond to a regular-tree grammar. This makes it easier for a programmer to determine which inclusions should hold. We also show how to extend our approach to refinements of datatypes which include functions and references, including datatypes with recursion in contravariant positions, unlike previous work on refinements. We can no longer formulate an inductive semantics in this case, but our experience suggests that

this extension validates those inclusions that are intuitively expected, and that forbidding such refinements would be limiting in practice.

For convenience, each datatype has a *default refinement* which has the same name as the datatype, and a datasort declaration that mirrors the datatype declaration. Thus, we can think of the above datatype declaration as also including the following declaration.

```
(*[ datasort bits = bnil | b0 of bits | b1 of bits ]*)
```

These datasorts make it easy to provide sorts that do no more checking than done by the type system. Further, our sort checker uses these default refinements when annotations are missing in positions required by our bidirectional algorithm, to ensure that we have a conservative extension of SML.

In the presence of datasort declarations, sort checking the pattern matching construct of ML presents a number of challenges. For example, consider the following code for a function which standardizes an arbitrary bit string by removing zeros at the end to satisfy the sort `nat`.

```
(*[ stdize <: bits -> nat ]*)
fun stdize bnil = bnil
  | stdize (b0 x) = (case stdize x
                       of bnil => bnil
                        | y => b0 y
                     )
  | stdize (b1 x) = b1 (stdize x)
```

The inner pattern match is the most interesting here. To check the branch "`y => b0 y`" it is critical that we take account of the sequential nature of ML pattern matching to determine that `y` can not be `bnil`. We achieve this by using a generalized form of sorts for patterns that accurately capture the values matched by previous patterns.

The example above is relatively simple; in the presence of nested patterns and products the situation is considerably more complicated, and generally requires "reasoning by case analysis" to check the body of each branch. When we perform such an analysis, we avoid the "splitting" into unions of basic components that was used in previous work on refinements. This is because it leads to a potential explosion in the case analysis that needs to be performed. We instead focus on *inversion principles* that are determined relatively directly from datasort declarations.

1.4.4 Extending to a sort checker for Standard ML

We tackled a number of other smaller challenges while extending sort checking to the full SML language. The following are some of the more notable related to the design of sort checking (as opposed to its implementation). When a required annotation is missing during sort checking of expressions, we use a default refinement that results in similar checking to that performed during type checking. We allow parameterized datasort declarations with variance annotations for each parameter. When we have a type sharing specification for types which have refinements, we also share all refinements according to their names. We allow the specification of refinements of opaque types in signatures, including specifications of the inclusions that should hold between them.

We also briefly mention some of the more notable challenges tackled while implementing our sort checker for SML. The implementation of the datasort inclusion algorithm uses a sophisticated form of memoization and other techniques in order to obtain acceptable performance. The implementation of the bidirectional checking algorithm uses a library of combinators for computations with backtracking and error messages which is designed to be efficient and allow a natural style of programming. The implementation of checking for pattern matching uses some important optimizations, such as avoiding redundancy during case analysis.

Our experiments with our implementation indicate that refinement-type checking is a practical and useful method for capturing and checking properties of real SML code. We found that the annotations required were very reasonable in most cases, and that the time taken to check was at most a few seconds. In general, the error messages produced were even more informative than those produced by an SML compiler: this is because bidirectional checking localizes the effect of errors better than type inference based on unification (as generally used for SML).

1.5 Introductory examples

We now show some additional examples. These use the type for bit strings and the sorts for the associated representation invariant for natural numbers that were introduced in the previous section.

```
datatype bits = bnil | b0 of bits | b1 of bits

(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat ]*)
```

We start with a very simple example: the constant four. As expected it has the sort `pos`. This sort could be inferred, so the annotation is not required, but serves as handy, mechanically checked documentation.

```
(*[ four <: pos ]*)
val four = b0 (b0 (b1 bnil))
```

In contrast, the following bit string consisting of three zeros is not even a natural number. The best sort it can be assigned is `bits`, i.e. the default refinement of the type bits. Again, this sort could be inferred.

```
(*[ zzz <: bits ]*)
val zzz = b0 (b0 (b0 bnil))
```

The next example is a function that increments the binary representation of a number.

```
(*[ inc <:  nat -> pos ]*)
fun inc bnil = b1 bnil
  | inc (b0 x) = b1 x
  | inc (b1 x) = b0 (inc x)
```

We remark that ascribing `inc <: nat -> nat` instead of `nat -> pos` would be insufficient: in order to determine that the last clause returns a valid natural number, we need to know that the result of the recursive call `inc x` is positive. This is reminiscent of the technique of strengthening an induction hypothesis which is commonly used in inductive proofs. Our experience indicates that it is not too hard for a programmer to determine when such stronger sorts are required for recursive calls, and in fact they must be aware of the corresponding properties in order to write the code correctly. In fact, using a sort checker actually helps a programmer to understand the properties their code, and thus allows correct code to be written more easily, particularly when the properties are complicated.

We further remark that subsorting allows us to derive additional sorts for `inc`, including `nat -> nat`, via inclusions such as the following.

```
nat -> pos  ≤  nat -> nat
nat -> pos  ≤  pos -> pos
```

We also remark that the sort we have ascribed will result in each call to `inc` having its argument checked against the sort `nat`, with an error message being produced if this fails. If some parts of the program were designed to manipulate natural numbers that temporarily violate the invariant (and perhaps later restore it using `stdize`), it might be appropriate to instead ascribe the following sort.

```
(*[ inc <: (nat -> pos) & (bits -> bits) ]*)
```

It seems reasonable to ascribe this sort, since the result returned is appropriate even when the invariant is violated.

Our next example is a function which adds together two binary natural numbers.

```
(*[ plus <:  (nat -> nat -> nat)
           & (nat -> pos -> pos) & (pos -> nat -> pos) ]*)
fun plus bnil n = n
  | plus m bnil = m
  | plus (b0 m) (b0 n) = b0 (plus m n)
  | plus (b0 m) (b1 n) = b1 (plus m n)
  | plus (b1 m) (b0 n) = b1 (plus m n)
  | plus (b1 m) (b1 n) = b0 (inc (plus m n))
```

Again, for this definition to sort-check we need to know that `inc <: nat -> pos`. We also need a stronger sort than `plus <: nat -> nat -> nat` for the recursive calls. For these, the following sort would have sufficed, but subsequent calls to `plus` would have less information about its behavior.

```
(*[ plus <: (nat -> nat -> nat) & (pos -> pos -> pos) ]*)
```

Next, we show an example of an error that is caught by sort checking.

```
(*[ double <: (nat -> nat) & (pos -> pos) ]*)
fun double n = b0 n
```

Our sort checker prints the following error message for this code.

```
fun double n = b0 n
               ^^^^
Sort mismatch: bits
    expecting: nat
```

This tells us that the expression `b0 n` has only the sort `bits`, but should have sort `nat`. To figure out why it does not have sort `nat`, we can look at the datasort declaration for `nat`: it specifies that `b0` must only be applied to arguments with sort `pos`. Indeed, `double bnil` evaluates to `b0 bnil` which is not a valid representation of zero. We are thus led towards an appropriate fix for this problem: we add an additional case for `bnil`, as follows.

```
(*[ double <: (nat -> nat) & (pos -> pos) ]*)
fun double bnil = bnil
  | double n = b0 n
```

We conclude this section with an example which shows some of the expressive power of sort checking with recursive datasort declarations. This example uses datasort declarations to capture the parity of a bit string, and verifies that a function that appends a one bit to the end of a bit string appropriately alters the parity.

```
(*[ datasort evPar = bnil | b0 of evPar | b1 of odPar
         and odPar =        b0 of odPar | b1 of evPar ]*)

(*[ append1 <: (evPar -> odPar) & (odPar -> evPar)    ]*)
fun append1 bnil = b1 bnil
  | append1 (b0 bs) = b0 (append1 bs)
  | append1 (b1 bs) = b1 (append1 bs)
```

The full power of sorts is not demonstrated by this small example. It it best demonstrated in the context of real code with all its complexities. See the experiments in Chapter 9 for examples of real code with sort annotations.

1.6 Related work

1.6.1 Previous work on refinement types for ML

Refinement types were introduced by Freeman and Pfenning [FP91]. We have already described the fundamentals of their work in Section 1.3, and considered the most relevant details in Section 1.4. In what follows we examine other relevant details, and comment on the relationship to our work.

Freeman and Pfenning originally extended a small subset of ML with recursively defined refinements of datatypes. They demonstrated the expressiveness of these refinements, and developed an algorithm for performing sort inference. This algorithm first constructs a lattice

of the defined refinements for each datatype, including intersections between them. Then the sort of an expression is inferred using a form of abstract interpretation (see the work of Cousot and Cousot [CC77]), which roughly "executes" the program, replacing the values of the language by the lattice elements. During sort inference, subsorting problems that arise are solved by a method that makes use of the constructed lattices for refinements for datatypes. Parametric polymorphism is included in their language, with the restriction that the only refinement of a type variable is a unique sort variable. They included a union operator on sorts as well as an intersection operator, although in later work unions were omitted.

In our work, we similarly construct lattices of refinements of datatypes, we use the same restriction on parametric polymorphism, and we include intersections but not unions.

Sort inference

Work on sorts in ML was continued by Freeman in his PhD thesis work [Fre94]. This work concentrated mostly on algorithms for inferring the sort of an expression, following the ideas in [FP91]. Efficiency turned out to be a major issue, since inferring the principal sort of a function essentially requires enumerating all refinements of the argument type. Additionally, inferring the sorts of recursive functions via abstract interpretation requires a fixed point calculation by iteration. Using sophisticated representations for sorts and memoization allowed an implementation to perform sort inference for many examples, though the largest such example was only about 50 lines of SML code. Further, the time taken for sort inference increased dramatically when more refinements were declared, and when higher-order functions were used. Our approach uses bidirectional checking to avoid these problem.

Language features

The work of Freeman focused on only a subset of ML, which did not include many features that are used frequently in real programs: these include pattern matching, exceptions, mutable references and modules. The lack of features involving effects means that some important aspects of Freeman's work do not scale to ML, and in our work we found it necessary to introduce a new form of intersection types that are sound in presence of effects. Also, the lack of pattern matching made realistic experiments particularly difficult: these had to be expanded to a single-level case construct by hand. Our work includes a formal treatment of a pragmatic approach to pattern matching. Further, we have extended sorts to the full SML language, allowing experimentation with real code.

Refinements of datatypes

Freeman also presented an algorithm for calculating lattices of refinements of constructed types, but this algorithm was not completely satisfactory, since it rejects some inclusions which should intuitively hold. We illustrate this with the following declarations. We recall the declarations of `nat` and `pos`.

```
(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat
         and zero = bnil
         and nat2 = bnil | b0 of pos
                         | b1 of zero | b1 of pos  ]*)
```

If we think of these as inductive definitions, then `nat` and `nat2` are equivalent. However, the algorithm of Freeman will only determine that `nat2` is included in `nat`, and rejects the inclusion in the other direction. In fact, there does not seem to be any obvious way to specify the results calculated by this algorithm, so the programmer must be familiar with the algorithm itself in order to predict which inclusions will be determined to hold.

The work of Freeman also includes an approach to refinements of parameterized datatypes. His approach includes covariant and contravariant parameters, with inference of variances via a fixed point calculation. Our approach to parameterized datatypes is mostly based on that of Freeman, although we require the programmer to declare the intended variance of parameters. This is generally a very modest requirement, and makes the variances obvious when reading the code. It also ensures that the variance is what the programmer expects, and allows a weaker variance to be declared than what would be inferred.

1.6.2 Refinement types for LF

Other work on refinement types includes that by Pfenning [Pfe93] on an extension of LF. This work is very similar in spirit to work on sorts for ML, but the technical and practical issues are very different. In this extension, the base sorts are "open ended", in the sense that they could be extended with further constructors by subsequent declarations. As a result subsorting is not be determined by analyzing the declarations as is done with datasort declarations in ML. Instead, the programmer directly declares the inclusions which hold. Also, in LF there is no sort inference problem to be solved, but only a sort checking problem, thus avoiding some of the efficiency problems with ML sort inference mentioned above. Furthermore, LF has no recursion or polymorphism, but does have dependent types, so sort checking requires very different techniques from those for ML.

In Chapter 2 and Chapter 3 we use a presentation of sorts for the simply-typed λ-calculus that is partially based on this work by Pfenning on sorts for LF. In particular, the base sorts are open ended, with their inclusions specified separately from their constructors. We do this in order to present sort checking in a very general context, so that the basic method can be extended to languages such as LF that are quite different from ML. In later chapters, we have elimination forms that require all constructors for a base sort to be included, so open-ended sorts are not appropriate.

1.6.3 Other forms of refinements of types

Other work has considered refinements of types that have quite a different character to ours. Some of these involve features other than intersections and unions, and some involve refinements of types quite different to datatypes. We broadly refer to these as *type refinements*, and reserve the term "refinement type" for type refinements involving only intersections, unions (in some cases) and constructor types (such as datatypes).

Hayashi [Hay94] uses a form of refinements to express specifications of programs, in a similar way to work on extracting programs from constructive logic proofs. Here, the ordinary type system is simply that of second-order λ-calculus, while the refinements include dependent sorts, as well as intersections and unions. Thus, in contrast to other work, extracting a program from a proof simply involves ignoring the level of refinements. As a result the underlying program is readily apparent during proof construction.

A similar line of work is that of Denney [Den98], who considers a very general form of refinements that include a full classical logic of properties. Denney also defines a semantics for refinements that involves a different equality for objects for each refinement of a type. This means that equality ignores irrelevant features of objects, such as the behavior of a function when applied to arguments that are ruled out by a particular refinement.

Xi and Pfenning [XP99, Xi98] consider a form of dependent refinements for ML which adds indices to types. Following our work, they use a bidirectional checking algorithm, but generate constraints involving the indices that must be solved. The main index domain that they focus on is integers with linear inequalities. Also following our work, they use a value restriction to achieve soundness in the presence of effects. They include examples that demonstrate that their refinements are very expressive, and allow many common program properties to be captured.

One of the examples presented by Xi [Xi98] involves an implementation of red-black trees, which was also the subject of one of our experiments. In Section 9.1 we report our results, and briefly compare with those obtained by Xi. We check only the coloring invariant of red-black trees, while Xi checks both the balancing invariant and the coloring invariant. One of our conclusions is that our refinements can be encoded using dependent refinements, at least in this case, but that the encoding is somewhat unnatural. This is important because unnatural encodings make writing and reading annotations difficult, and also result in the errors generated by a type checker being much harder to interpret.

Dunfield and Pfenning [Dun02, DP03, DP04] consider a variety of extensions to the refinements and algorithms presented in this dissertation. Firstly, they consider the combination with indexed dependent refinements. This allows the natural expression of properties using the features of both systems. In the case of the red-black tree example, it would allow the coloring invariant to be naturally expressed using datasorts and intersections, and it would allow the balancing invariant to be naturally expressed via indices. Dunfield and Pfenning additionally show how to extend our value restriction on intersections to unions via a restriction on the elimination rule to evaluation contexts, which is roughly the dual of our restriction. They also and extend our bidirectional approach to a tridirectional one: the third direction follows to order of evaluation, and is required for the evaluation contexts. Further, they consider the slightly more general situation where there is no refinement restriction, i.e. there is only a single level a of types that includes intersections, unions and dependent types (although later, in Chapter 4, we will also consider a language without the refinement restriction). Finally, they generalize annotations to *contextual annotations* which elegantly handle to situation where different annotations are required depending on the sorts assigned to the variables in the context.

Mandelbaum, Walker and Harper[MWH03] show how to formulate refinements that capture imperative properties of code. Their approach includes a linear logic that allows local reasoning about properties that depend upon the current state. They also follow our work in that they use a bidirectional checking algorithm, which in their case may require proving linear logic theorems. Their work does not need a value restriction, since they build on a language which syntactically

distinguishes potentially effectful expressions from effect-free terms, following the reformulation of the computational meta-language of Moggi [Mog91] by Pfenning and Davies[PD01].

1.6.4 Program analysis via types

Other work on capturing detailed properties of programs using types includes many uses of types to specify program analyses. In this case, the types are generally used only as an internal mechanism for formulating an algorithm: they are not exposed to the programmer in the way we would expect for a type system.

For example, Nielson and Nielson [NN92] and Gomard and Jones [GJ91] have presented systems for binding-time analysis using types that are annotated with binding times.[1]

Another example is the work of Palsberg and O'Keefe [PO95], who present a type system which is equivalent to flow analysis. This line of work was continued by Palsberg and Pavlopoulou[PP01] who show that polyvariant flow can be captured using a particular type system that includes intersection and union types.

In similar work, Naik and Palsberg [Nai04][NP04] have demonstrated that a form of model checking for imperative safety properties is equivalent to a particular type system with intersection and union types.

Other examples in this vein are reported in [NS95]. A general framework for formulating program analyses as annotated type systems has been proposed by Solberg [Sol95].

1.6.5 Other forms of more detailed types

Foster, Fähndrich and Aiken [FFA99] have proposed *qualified types* as a method for checking more detailed properties of programs. Their motivations are very similar to ours, and one view might consider them to be a form of refinements. Many of the details of their system are different from ours however: e.g. they have a form of constrained polymorphic types, but no intersections. Also, the practical focus of their work is on C and C-like languages. Foster, Terauchi and Aiken [FTA02] have considered a form of qualified types that are flow sensitive, and hence are quite different from our refinements.

Barthe and Frade[BF99] have considered languages with *constructor subtyping*, which is subtyping between datatypes based on the absence of certain constructors. This is very similar to our datasort declarations. One difference in their setting is that they don't have an explicit refinement restriction: the subtyping is between types rather than between refinements of them. However, the types allowed for constructors are restricted in a way that is seems related to our refinement restriction. Also, they do not include intersection types: instead they allow only a restricted form of overloading.

Other relevant work on more detailed types includes the use of *phantom types* to increase the amount of checking done by the type system in languages with polymorphism and parameterized type constructors. This can be done without modifying the language itself: instead extra "phantom" type parameters are added to types. It is unclear who first used this technique: it appears to have been used by a number of people for some time without being reported in the literature.

[1] A consideration of the proper relationship between these type systems and refinement types ultimately led to very fruitful work in collaboration with Pfenning [Dav96, DP01, PD01], but which is not directly related to refinements since binding times may affect the semantics of programs.

Fluet and Pucella [FP02] have shown that phantom types can be used to encode any finite subtyping hierarchy within a Hindley-Milner[Hin69, Mil78] type system. The technique generally does not require the expressions of a program to be modified very much, although in some cases explicit coercions are required. However, the encoding does seem to be somewhat awkward when writing types, and particularly awkward when type errors are reported. They also show how subtypes of ML datatypes can be encoded. This allows some of the properties that we check using sorts to be checked instead using the ordinary type system of ML. However, using this technique seems much more awkward than using our sorts. It also does not extend to intersections, which are critical for most of our complicated examples involving recursion.

This technique also does not result in the desired exhaustiveness checks being performed for pattern matches. This can be remedied by extending the underlying language with the *guarded recursive datatype constructors* of Xi, Chen and Chen [XCC03], or the *first-class phantom types* of Cheney and Hinze [CH03], which are similar. However, this loses one of the main advantages of phantom types: that they can be used in existing languages without changing them.

1.6.6 Soft typing

Soft typing aims to bring some of the benefits of statically-typed languages to dynamically typed ones, and was first considered by Reynolds [Rey69], although the term "soft typing" is due to Cartwright and Fagan [CF91]. This is done using analyses or type systems which do not require any type declarations by the programmer, and then automatically inserting dynamic type checks based on the results to guarantee that the resulting program cannot raise a runtime error. Programmers can use the dynamic type checks to help locate possible errors, and some work has been done by Flanagan et al. [FFK+96] to design a sophisticated interface to help in this process. However, even with such an interface the process is tedious enough that programmers are likely to make errors occasionally, particularly when a program is modified many times, and these errors would have unfortunate consequences.

Generally these systems include some form of subtyping, and can express inclusions that have some similarities to inclusion between our datasorts. For example, the system of Wright and Cartwright[WC94] includes union types, recursive types and types for applications of constructors. Additionally, the type systems used for soft typing often capture quite accurate invariants of programs, so one might wonder whether they could be used for the kind of invariants that our sorts are designed to capture.

It seems that some of these invariants can be captured by soft-typing systems, but that many of those in which we are most interested can not. In particular, many invariants for recursive functions that can be captured using recursive datasorts are are not accurately reflected in the types inferred by soft typing systems. A simple example is the `append1` function for bit strings presented in Section 1.5, and repeated here.

```
(*[ datasort evPar = bnil | b0 of evPar | b1 of odPar
         and odPar =        b0 of odPar | b1 of evPar ]*)

(*[ append1 <: (evPar -> odPar) & (odPar -> evPar)    ]*)
fun append1 bnil = b1 bnil
  | append1 (b0 bs) = b0 (append_one bs)
  | append1 (b1 bs) = b1 (append_one bs)
```

Soft-typing systems generally do not infer a type that captures the invariant expressed by the sort annotation for `append1`. We focus on the soft-typing system of Aiken, Wimmers and Lakshman [AWL94], which appears to be one of the most accurate. This system combines the conditional types of Reynolds [Rey69] with the constrained types of Mitchell [Mit84], and additionally includes restricted forms of unions and intersections. If we apply this system to code corresponding to that for `append_one`, we obtain the following constrained type.

$$\begin{array}{l} \forall\alpha.\alpha \to \beta \\ \quad \text{where } \alpha \;\leq\; \texttt{bnil} \cup \texttt{b0}(\alpha) \cup \texttt{b1}(\alpha) \\ \qquad\quad \beta \;=\; (\texttt{b1}(\texttt{bnil}) \;?\; (\alpha \cap \texttt{bnil})) \\ \qquad\qquad \cup \quad (\texttt{b0}(\beta) \;?\; (\alpha \cap \texttt{b0}(1))) \\ \qquad\qquad \cup \quad (\texttt{b1}(\beta) \;?\; (\alpha \cap \texttt{b1}(1))) \end{array}$$

This type is polymorphic in the argument type α and result type β, but the instances chosen must satisfy the two constraints. Roughly, the first constraint means that α must be a subtype of `bits` that contains the bit string x if it contains either `b0` x or `b1` x. The second constraint means that if the instance for α has a non-nil intersection with the type containing only `bnil`, then the result type β contains `b1(bnil)`, and if α contains any value beginning with `b0` (1 is an all-inclusive type) then β contains `b0`(x) for each x in β, and if α contains any value beginning with `b1` then β contains `b1`(x) for each x in β.

This type contains a lot of information, but it does not capture the desired invariants. In particular, `evPar` and `odPar` do not correspond to valid instantiations of α, since they violate the first constraint. In fact, the only supertype of either of these types that would be a valid instantiation of α is the one which corresponds to the whole of the type `bits`.

A closer analysis of this result indicates that the problem is the lack of polymorphic recursion in the system of Aiken, Wimmers and Lakshman. This suggests that polymorphic recursion would be a very useful addition to this system, though it seems very unlikely that type inference would be decidable for such a system. It is also unclear whether a workable system could be designed without full inference of types.

1.6.7 Intersection types

The inclusion of an intersection operator allows sorts to express very precise program properties by combining many pieces of information into a single sort. Such intersection operators have been extensively studied in the context of type assignment systems for λ-calculi, and were originally introduced by Coppo, Dezani-Ciancaglini and Venneri [CDV81]. An important property of most of these systems that a λ-term can be assigned a type exactly when it has a normal form. In particular, this is true for the simplest such system, which includes only intersection types, function types, and a single base type. Thus, intersection types add considerable expressive power in this context. Since the existence of a normal form is undecidable, type assignment is also undecidable in this context.

The use of intersection types in programming languages was first proposed by Reynolds who used them in the language Forsythe [Rey81, Rey96]. In Forsythe intersections serve several purposes, including: to express overloading of arithmetic functions, to express polymorphism (in the absence of parametric polymorphism), and to allow a flexible approach to records with subtyping. However, they are generally not used to represent detailed program properties, as

we do in our work with sorts. This is mostly due to Forsythe lacking types that correspond to ML datatypes.

Reynolds [Rey96] has presented a type checking algorithm for Forsythe which allows the types of some variables to be omitted. This algorithm uses a generalization of a bidirectional approach, although it has some significant differences from our bidirectional algorithm for sort checking. The main reason for this is that the Forsythe algorithm is based on an approach that achieves improved efficiency by making assumptions about the forms of inclusions that may hold between base types. In particular, the algorithm is incomplete when there are base types such that $a_1 \;\&\; a_2 \leq b$ holds but neither $a_1 \leq b$ nor $a_2 \leq b$ hold. The subtyping algorithm for Forsythe outlined by Reynolds makes a similar assumption, and thus also is incomplete in the presence of such base types. These incompletenesses do not seem to have been observed previously. In Forsythe they are not an issue, since the set of base types is fixed, and the algorithm is complete for these. We discuss this further in Section 2.9.

Another difference between our sort checking algorithm and the type checking algorithm of Reynolds is that we distinguish only inferable and checkable expressions, while Reynolds allows fewer type annotations in some cases by also distinguishing expressions for "functions which can be checked against types with the first n arguments unknown" for each n. We prefer our distinction because it is simpler to describe, and our experience indicates that it is sufficient in practice. Further, it is fundamentally related to the syntactic definition of normal forms for the λ-calculus. If future experience with programming with sorts reveals cases where excessive annotations are required with our approach, then we may consider an extension along the lines of Reynolds.

One important technique that we use is directly based on the work of Reynolds: our sort annotations include a list of alternative sorts. This is necessary in the presence of intersections, since each expression may need to be checked against many different sorts. In the work of Reynolds, annotations are placed on bound variables, and multiple alternatives are frequently used on function arguments to produce sorts involving intersections, e.g., $\lambda x{:}A{,}B\,.\,x$ would be assigned $(A \to A)\,\&\,(B \to B)$. While we borrow the technique directly from Reynolds, we differ in that our annotations are on expressions instead of bound variables. This means that intersections sorts can be directly assigned to functions, and multiple alternatives are generally only required when an annotated expression is checked under multiple contexts, and different sorts must be assigned depending on the context. Experience so far indicates that this situation is very rare.

While allowing multiple alternative annotations for an expression is sufficient to allow different sorts to be assigned under different contexts, this solution is still a little unsatisfactory. Each time the expression is checked under a context, there is generally only one alternative that is intended by the programmer for that context, but this information is not expressed by the annotation, and all of the alternatives must be tried. This is not only inefficient, it also makes the code harder to read because the dependence on the context is not made explicit. This is solved by a natural and elegant generalization of this technique by Dunfield and Pfenning [DP04] to contextual annotations, which allow each alternative to be preceded by a context that specifies the situations where the alternative applies. In their case, this generalization is essential due to the presence of indexed dependent types. In our case contextual annotations may be desirable, but since multiple alternatives are so rare, it seems reasonable to omit them until a practical need for them arises.

An important result proved by Reynolds [Rey96] is that Forsythe type checking is PSPACE-hard, even when the types of all variables are declared. The proof involves translating quantified boolean expressions into Forsythe programs which type check exactly when the boolean expression is true. Exactly the same technique can be used to show that ML sort checking is PSPACE-hard. In practical terms this means that it is possible for sort checking not to terminate within a reasonable amount of time. However, experience so far indicates that sort checking is likely to be efficient enough for those programs encountered in practice.

Other relevant work on intersection types includes that of Pierce [Pie91, Pie97], who considers the combination of intersection types and bounded polymorphism. Pierce includes a formal account of algorithms for subtyping and type checking with intersection types. These algorithms are based on a similar approach to those of Reynolds.

1.6.8 Regular tree types

Part of the original inspiration for adding sorts to ML was the use of regular tree types in logic programming. These types represent subsets of the Herbrand universe of a logic programming language. Thus, it is very natural for regular tree types to have a subtype ordering corresponding to inclusion of the sets they represent. Regular tree types are defined by regular term grammars, which allow this ordering to be computed relatively efficiently, while also being expressive enough to specify appropriate sets in most cases. This also allows intersections, unions and complements of regular tree types to be computed.

Much of the work on regular tree types, such as [Mis84], has restricted the grammars further so that they are tuple distributive, which means that if $f(A_1, A_2)$ and $f(B_1, B_2)$ are in the set generated by the grammar, then so must $f(A_1, B_2)$ and $f(A_2, B_1)$. This allows regular tree grammars to be treated similarly to ordinary regular grammars, for which good algorithms are well known, but it also reduces their expressiveness considerably. In our case, this restriction is inappropriate because it seems unnatural and would severely limit the expressiveness of datasort declarations in practice. In particular, in our experiments many of the situations where sorts were most useful involved non-distributive tuples.

General regular tree grammars are closely related to tree automata, which have been the subject of much recent research. Common et. al. [CDG$^+$] present a survey of the field. In our case we are most interested in the problem of checking inclusion between grammars. This problem can be shown to be EXPTIME-hard by an easy reduction from the problem of inequivalence of finite tree automata, which was shown to be EXPTIME-complete by Seidl [Sei90].

Despite this, algorithms for this problem have been proposed by a number of researchers. Implementations based on these algorithms appear to achieve acceptable performance, at least for those instances that arise in systems which use types based on grammars. The first such algorithm was proposed by Aiken and Murphy [AM91], although they found that they found it necessary to make some approximations to achieve acceptable performance (in their context such approximations are acceptable).

Hosoya, Vouillon and Pierce [HVP00] have presented a similar algorithm to Aiken and Murphy, and using some sophisticated implementation techniques they appear to obtain acceptable performance without approximations. A quite different algorithm has been proposed by Benzaken, Castagna, and Frisch [BCF03], and a similarly sophisticated implementation appears to obtain acceptable performance.

The algorithm we present in Chapter 5 is similar to that of Hosoya, Vouillon and Pierce. Our implementation similarly required sophisticated techniques, and has some similarities with the other implementations mentioned above. See Section 8.4 for a detailed comparison.

The general form of our datasort declarations included in our extension of SML allows sorts to be defined by regular tree grammars, but additionally allows sort parameters (and occurrences of function and reference sorts). Skalka [Ska97] formally proved the soundness and completeness an algorithm for determining emptiness of such parameterized datasort definitions. This algorithm contains many of the essential ideas required to construct an inclusion algorithm for parameterized datasort declarations: all that would be required is an algorithm for calculating differences between sorts. Our inclusion algorithm is, in a sense, incomplete for such parameterized definitions. In Section 7.4.3 we discuss this issue further, and include a detailed comparison with the work of Skalka.

Aiken and Wimmers [AW93] have considered the problem of solving type inclusion constraints, which are essentially set constraints with partial function types added. They use a denotational semantics of types based on ideal models [MPS86] to define a natural notion of inclusion that allows recursively defined types. Unfortunately their algorithm requires restrictions on occurrences of intersections and unions, and these restrictions do not seem appropriate for sort definitions.

The algorithm of Benzaken, Castagna, and Frisch[BCF03] mentioned above does include both intersection types and partial function types. However, it does not directly apply to our situation, because we have different inclusions due to the presence of effects and our value restriction on intersections.

1.7 Organization of this dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 focuses on sorts and sort checking for a λ-calculus using the standard intersection type rules.

Chapter 3 focuses on sorts and sort checking for a λ-calculus using a value restriction on intersections, and a modified subsorting for intersections so that the language is suitable for extension with effects.

Chapter 4 demonstrates that our value restriction and modified subsorting result in a system that is sound in presence of effects by adding mutable references and an example datatype.

Chapter 5 presents our approach to datasort declarations, including our algorithm for comparing them for inclusion.

Chapter 6 considers sort checking in the presence of datasort declarations, including pattern matching.

Chapter 7 describes the design of an extension of our approach to the full set of features in Standard ML.

Chapter 8 describes our implementation of a sort checker for SML, based on the design and algorithms in previous chapters.

Chapter 9 describes some experiments with sort checking of real SML code using our implementation.

Chapter 2

Sort checking with standard intersection types

This chapter focuses on bidirectional sort checking with the standard intersection type rules (i.e., without a value restriction and with distributivity). Intersection types with these rules have been extensively studied in the context of λ-calculi (see e.g. [CDV81]) and in the context of programming languages (e.g. [Rey91][Rey96]). Refinement types with these rules have been studied previously in two contexts: an extension of the programming language ML [FP91, Fre94], and an extension of the logical framework LF [Pfe93].

The sorts in this chapter are not suitable for languages with call-by-value effects (see Section 1.4.2 and the introduction to Chapter 3). Since the practical focus of this work is sort checking for SML, which includes such effects, the sorts in this chapter do not play a central role in the remainder of this dissertation. Thus, this chapter should be considered secondary to Chapter 3, which presents sorts which are suitable for extension with call-by-value effects, and includes the algorithm that forms the core of our implementation of sort checking for SML.

We have chosen to include this chapter in this dissertation mostly because it provides an intermediate point when comparing our main sort checking algorithm with previous work. The algorithm in this chapter follows a similar approach to that in Chapter 3, and is quite different to previous bidirectional algorithms for intersection types, such as those of Reynolds [Rey96] and Pierce [Pie97]. This is because these previous algorithms require restrictions on what kinds of inclusions are allowed between base types, and these restrictions are not appropriate for sorts. Thus, the algorithm in this chapter may also be of interest to those seeking a general checking algorithm for standard intersection types which makes as few assumptions as possible.

We also take the opportunity in this chapter to demonstrate a slightly more abstract presentation than in Chapter 3, in particular in the way that refinements of base types are declared in signatures. This results in the algorithms in this chapter being slightly further from the actual implementation. It is our intention that the more abstract presentation be considered as a possible "front-end" to the presentation of Chapter 3 (via a correspondence between abstract signatures and finite lattices, which will be demonstrated formally in Section 2.6). Similarly, the more efficient approach in Chapter 3 can be considered to be an implementation technique for the sorts in this chapter. Since the value restriction and distributivity do not affect the structure of refinements of base types, this is an orthogonal concern.

Because the development in this chapter is quite similar to that in Chapter 3, and the sorts

in this chapter are not as central, the presentation in this chapter is deliberately terse. The development in Chapter 3 is explained in more detail and we include forward pointers where appropriate.

The remainder of this chapter is structured as follows. Section 2.1 presents the syntax of a simply-typed λ-calculus, parameterized by a signature containing declarations of base types, base sorts, and constants, and also declaring inclusions between refinements of base types. Section 2.2 presents validity judgments for signatures, types, refinements, contexts and terms. Terms are judged to be valid via the standard typing rules, but we do not include sort assignment rules: we do not consider that the purpose of sort assignment is to determine the validity of terms. Section 2.3 presents the standard notion of reduction. Section 2.4 presents declarative subsorting rules, based on the standard rules for intersection types. Section 2.5 considers equivalence of sorts, and shows that the equivalence classes of refinements of a type form a finite lattice. Section 2.6 shows that our declarations of inclusions in signatures are equivalent to allowing arbitrary finite lattices of refinements of base types (as is done in Chapter 3). Section 2.7 briefly considers algorithmic base subsorting. Section 2.8 presents declarative sort assignment rules. Section 2.9 presents our algorithm for subsorting and an outline of the proof of its correctness. Section 2.10 presents our bidirectional sort checking algorithm, and an outline of the proof of its correctness. Section 2.11 demonstrates that terms can always be annotated appropriately as required by our sort checking algorithm.

2.1 Syntax

We now present the syntax of our simply-typed λ-calculus with sorts, which we call $\lambda^{\neg\&}$. We follow Pfenning's presentation of the simply-typed λ-calculus (see Section 6 of [Pfe01b] or Chapter 3 of [Pfe05]). We add sorts, placing types and sorts in separate syntactic classes, following Freeman [Fre94], but differing from Pfenning's presentation of refinement types for LF [Pfe93]. We also include zero-ary intersections, or "top" sorts, using the syntax $\top^A$, thus making the type that is being refined inherent in the syntax of the sort. To avoid clutter, we often omit the type A and simply write $\top$ when the type can be reconstructed from the context or is of little interest. These top sorts are somewhat different from those considered by Freeman [Fre94], which are not represented using a special syntactic form, but are instead constructed in a way that ensures they are maximum elements.

We include signatures with declarations of both sort constants and subsorting relationships, following Pfenning [Pfe93], but differing from Freeman [Fre94]. We allow subsorting declarations of the form $R \overset{a}{\leq} S$ where R and S are refinements of a type constant a. This extends the declarations in [Pfe93] which have the form $r \leq s$. The extension is necessary so that all finite lattices of base sorts can be declared (see Section 2.6).

We use a, b for type constants, r, s for sort constants, c for term constants and x for term variables.

$$
\begin{array}{rrl}
\text{Types} & A, B & ::= a \mid A_1 \to A_2 \\
\text{Type Contexts} & \Gamma & ::= \cdot \mid \Gamma, x{:}A \\
\text{Terms} & M, N & ::= c \mid x \mid \lambda x{:}A.M \mid M_1\, M_2 \\
\text{Sorts} & R, S & ::= r \mid R_1 \to R_2 \mid R_1 \,\&\, R_2 \mid \top^A \\
\text{Sort Contexts} & \Delta & ::= \cdot \mid \Delta, x{\in}R \\
\text{Declarations} & D & ::= a{:}\mathsf{type} \mid r \sqsubset a \mid R \overset{a}{\leq} S \mid c{:}A \mid c{\in}R \\
\text{Signatures} & \Sigma & ::= \cdot \mid \Sigma, D
\end{array}
$$

We write $\{N/x\}M$ for the result of substituting N for x in M, renaming bound variables as necessary to avoid the capture of free variables in N. We use the notation $\mathcal{D} :: J$ to indicate that $\mathcal{D}$ is a derivation of judgment J.

We require variables to appear at most once in a type or sort context. Similarly, we require signatures to include at most one declaration $a{:}\mathit{type}$ for each a, at most one declaration $r \sqsubset a$ for each r, and at most one declaration $c{:}A$ and one declaration $c{\in}R$ for each c.

The intention of our signatures is that our simply-typed λ-calculus is parameterized with respect to a signature that declares:

- a set of base types $a{:}\mathsf{type}$
- a set of base sorts refining each base type $r \sqsubset a$ which satisfy inclusions $R \overset{a}{\leq} S$ (e.g. $r_1 \,\&\, r_2 \overset{a}{\leq} r_3$)
- a set of constants each inhabiting a type and refinement of that type $c{:}A, c{\in}R$

There appear to be a number of reasonable alternatives to our formulation of signatures.

- We could have separate signatures for type and sort level declarations.
- We could remove declarations of the form $c{:}A$ and instead make $c{:}A$ a consequence of $c{\in}R$ when $R \sqsubset A$.
- We could allow $c{\in}R_1$ and $c{\in}R_2$ to appear in the same signature (which would be equivalent to $c{\in}R_1 \,\&\, R_2$) as is done in Pfenning's refinements for LF [Pfe93].

These alternatives appear to result in only cosmetic differences.

2.2 Validity judgments

The validity judgments for $\lambda^{\to\&}$ extend those of Pfenning [Pfe01b, Pfe05]. In particular, the validity judgment for signatures is extended to the new forms of declarations $r \sqsubset a$, $c{\in}R$ and $R \overset{a}{\leq} S$. We have a new judgment that judges the validity of a sort as a refinement of a particular valid type. This judgment requires the type to be valid: if it is not we do not consider the judgment to be well formed. We use similar well-formedness restrictions for many judgments in this dissertation: without them many inference rules would require additional premises that are not directly related to the rule, and there would be a distracting amount

of clutter in derivations. We only consider well-formed instances of judgments: if an instance is not well formed it has no meaning, and it makes no sense to consider whether it holds or whether there is a derivation of it.

$\vdash \Sigma\ \mathit{Sig}$	Σ is a valid signature
$\vdash_\Sigma A : \mathsf{type}$	A is a valid type
$\vdash_\Sigma R \sqsubset A$	R is a valid refinement of type A. ("R refines A")

Valid signatures

$$\frac{}{\vdash \cdot\ \mathit{Sig}}\ \mathsf{sigemp} \qquad \frac{\vdash \Sigma\ \mathit{Sig}}{\vdash \Sigma, a{:}\mathsf{type}\ \mathit{Sig}}\ \mathsf{sigtyp} \qquad \frac{\vdash \Sigma\ \mathit{Sig} \quad \vdash_\Sigma A : \mathsf{type}}{\vdash \Sigma, c{:}A\ \mathit{Sig}}\ \mathsf{sigobj}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \quad \vdash_\Sigma a : \mathsf{type}}{\vdash \Sigma, (r \sqsubset a)\ \mathit{Sig}}\ \mathsf{sigsrt} \qquad \frac{\vdash \Sigma\ \mathit{Sig} \quad c{:}A \text{ in } \Sigma \quad \vdash_\Sigma R \sqsubset A}{\vdash \Sigma, c{\in}R\ \mathit{Sig}}\ \mathsf{sigobjsrt}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \quad \vdash_\Sigma R \sqsubset a \quad \vdash_\Sigma S \sqsubset a}{\vdash \Sigma, (R \overset{a}{\leq} S)\ \mathit{Sig}}\ \mathsf{sigsub}$$

Valid types

$$\frac{a{:}\mathsf{type} \text{ in } \Sigma}{\vdash_\Sigma a : \mathsf{type}}\ \mathsf{typcon} \qquad \frac{\vdash_\Sigma A : \mathsf{type} \qquad \vdash_\Sigma B : \mathsf{type}}{\vdash_\Sigma A \to B : \mathsf{type}}\ \mathsf{typarrow}$$

Valid refinements

$$\frac{r \sqsubset a \text{ in } \Sigma}{\vdash_\Sigma r \sqsubset a}\ \mathsf{srtcon} \qquad \frac{\vdash_\Sigma R \sqsubset A \qquad \vdash_\Sigma S \sqsubset B}{\vdash_\Sigma R \to S \sqsubset A \to B}\ \mathsf{srtarrow}$$

$$\frac{\vdash_\Sigma R_1 \sqsubset A \qquad \vdash_\Sigma R_2 \sqsubset A}{\vdash_\Sigma R_1 \,\&\, R_2 \sqsubset A}\ \mathsf{srtinter} \qquad \frac{}{\vdash_\Sigma \top^A \sqsubset A}\ \mathsf{srttop}$$

A direct consequence of these definitions is that every sort refines at most one type.

We say that a sort R is *well-formed* if it refines some type A. In what follows, we are only interested in well-formed sorts, and so when we use the term "sort" we implicitly mean "well-formed sort". We say that two sorts are *compatible* if they refine the same type.

As an example of a valid signature, the following represents the types for the example in the introduction involving bit strings: we have a type a_{bits} with positive and natural numbers as refinements r_{nat}, r_{pos} as well as a refinement r_{bits} that includes all bit strings. We include the inclusions $r_{\mathsf{pos}} \leq r_{\mathsf{nat}} \leq r_{\mathsf{bits}}$.

$$
\begin{array}{lll}
a_{\mathsf{bits}} & : & \mathsf{type}, \\
r_{\mathsf{nat}} & \sqsubset & a_{\mathsf{bits}}, \\
r_{\mathsf{pos}} & \sqsubset & a_{\mathsf{bits}}, \\
r_{\mathsf{bits}} & \sqsubset & a_{\mathsf{bits}}, \\
c_{\mathsf{bnil}} & : & a_{\mathsf{bits}}, \\
c_{\mathsf{b0}} & : & a_{\mathsf{bits}} \to a_{\mathsf{bits}}, \\
c_{\mathsf{b1}} & : & a_{\mathsf{bits}} \to a_{\mathsf{bits}},
\end{array}
\qquad
\begin{array}{lll}
r_{\mathsf{pos}} & \overset{a_{\mathsf{bits}}}{\leq} & r_{\mathsf{nat}}, \\
r_{\mathsf{nat}} & \overset{a_{\mathsf{bits}}}{\leq} & r_{\mathsf{bits}}, \\
c_{\mathsf{bnil}} & \in & r_{\mathsf{nat}}, \\
c_{\mathsf{b0}} & \in & (r_{\mathsf{bits}} \to r_{\mathsf{bits}}) \;\&\; (r_{\mathsf{pos}} \to r_{\mathsf{pos}}), \\
c_{\mathsf{b1}} & \in & (r_{\mathsf{bits}} \to r_{\mathsf{bits}}) \;\&\; (r_{\mathsf{nat}} \to r_{\mathsf{pos}})
\end{array}
$$

The validity judgments for type contexts and terms are completely standard. We add a judgment for validity of sort contexts refining a valid type context. Each of these judgments is with respect to a valid signature Σ. The validity judgment for terms requires a valid type context and a valid type in order to be well formed. The validity judgment for sort contexts requires a valid type context.

$\vdash_\Sigma \Gamma\ \mathit{Ctx}$	Γ is a valid context
$\Gamma \vdash_\Sigma M : A$	M is a valid term of type A in valid type context Γ
$\vdash_\Sigma \Delta \sqsubset \Gamma$	Δ is a valid refinement of valid type context Γ.

Valid type contexts

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{Ctx}}\ \mathsf{ctxemp} \qquad \frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A\ \mathit{Ctx}}\ \mathsf{ctxobj}$$

Valid terms

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash_\Sigma c : A}\ \mathsf{objcon} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : A}\ \mathsf{objvar}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.M : A \to B}\ \mathsf{objlam} \qquad \frac{\Gamma \vdash_\Sigma M : A \to B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : B}\ \mathsf{objapp}$$

Valid sort contexts

$$\frac{}{\vdash_\Sigma \cdot \sqsubset \cdot}\ \mathsf{sctxemp} \qquad \frac{\vdash_\Sigma \Delta \sqsubset \Gamma \qquad \vdash_\Sigma R \sqsubset A}{\vdash_\Sigma (\Delta, x{\in}R) \sqsubset (\Gamma, x{:}A)}\ \mathsf{sctxobj}$$

We do not include a judgment for validity of terms with respect to sorts here because we consider that this is not the purpose of sorts. Instead we later present a sort *assignment* judgment that assigns sorts to terms that are already judged to be valid using the typing judgment. This reflects our view of the different purposes of types and sorts: types are required for validity, following the style of Church [Chu40], while sorts assign properties to terms that are already known to be valid, roughly in the style in Curry [Cur34].

2.3 Reduction

We have the standard β-reduction rule for terms.

$$\frac{}{(\lambda x{:}A.M)\, N \mapsto \{N/x\}M}\ \beta$$

We also have the compositional rules:

$$\frac{M \mapsto N}{\lambda x{:}A.M \mapsto \lambda x{:}A.N}\ \mathsf{reduce\_lam}$$

$$\frac{M \mapsto M'}{M\, N \mapsto M'\, N}\ \mathsf{reduce\_app1} \qquad \frac{N \mapsto N'}{M\, N \mapsto M\, N'}\ \mathsf{reduce\_app2}$$

So far $\lambda^{\rightarrow\&}$ is just the simply-typed λ-calculus in the style of Church. We omit the proofs of standard results such as subject reduction with respect to types. Our main interest is in the refinements of the standard types.

2.4 Declarative subsorting

The subsorting judgment has the following form, where R and S must be compatible sorts in order for the judgment to be well formed.

$\vdash_\Sigma R \leq S \quad R$ is a subsort of S

The subsorting rules are standard (see e.g. [Rey91]). The signature Σ is fixed throughout these rules, and we omit the $\vdash_\Sigma$ for brevity here and as appropriate in the remainder of this chapter.

$$\frac{R \overset{a}{\leq} S \text{ in } \Sigma}{R \leq S}\ \mathsf{sub\_base} \qquad \frac{R' \leq R \qquad S \leq S'}{R \to S \leq R' \to S'}\ \mathsf{sub\_arrow}$$

$$\frac{}{R \leq R}\ \mathsf{sub\_reflex} \qquad \frac{R_1 \leq R_2 \qquad R_2 \leq R_3}{R_1 \leq R_3}\ \mathsf{sub\_trans}$$

$$\frac{}{R \,\&\, S \leq R}\ \mathsf{sub\_\&left1} \qquad \frac{}{R \,\&\, S \leq S}\ \mathsf{sub\_\&left2}$$

$$\frac{R \leq S \qquad R \leq S'}{R \leq S \,\&\, S'}\ \mathsf{sub\_\&right} \qquad \frac{}{R \leq \top^A}\ \mathsf{sub\_Tright}$$

$$\frac{}{(R \to S) \,\&\, (R \to S') \leq R \to (S \,\&\, S')}\ \mathsf{sub\_\&dist} \qquad \frac{}{\top^{A \to B} \leq R \to \top^B}\ \mathsf{sub\_Tdist}$$

The requirement that R and S be compatible in order for $R \leq S$ to be well formed allows us to omit certain premises that would otherwise need to be explicitly included in the rules. For example, in the rule sub_Tdist, we must have $R \to \top^B \sqsubset A \to B$, and hence $R \sqsubset A$.

2.5 Sort equivalence and finiteness

If $R \leq S$ and $S \leq R$ then we say R and S are *equivalent* sorts. We use the notation $R \cong S$ for this relation, which is an equivalence relation: it is reflexive by rule sub_reflex (in each direction) and transitive by rule sub_trans (in each direction).

Our main motivation for introducing this definition is that there are only a finite number of refinements of each type modulo sort equivalence, which can be proved by induction on the structure of the type refined by R and S. We omit a formal proof here. Section 3.6 includes a formal proof of a finiteness of refinements theorem (Theorem 3.6.3) in the absence of the rule sub_dist. Since the removal of this rule can only result in more distinct refinements, this proof can easily be adapted to the situation in this chapter. Also, Freeman [Fre94] has presented a finiteness theorem for a system of sorts with distributivity.

We now show that & is associative and commutative. We start with the following definitions, and some basic lemmas.

Definition 2.5.1 (Conjunct, Basic conjunct)
We say that R is a conjunct *of S if one of the following holds.*

- $S = R$
- $S = S_1 \,\&\, S_2$ *and (inductively) R is a conjunct of either S_1 or S_2.*

When R is additionally not an intersection or $\top^A$, we say that R is a basic conjunct *of S.*

Lemma 2.5.2 (Conjunct inclusion)
If S is a conjunct of R then $R \leq S$.

Proof: By induction on R.

Case: $R = S$. Then $S \leq S$ by rule sub_reflex.

Case: $R = R_1 \& R_2$ with S a conjunct of R_1.

Then $R_1 \leq S$ by the induction hypothesis,
and $R_1 \& R_2 \leq R_1$ by rule sub_&left1,
so $R_1 \& R_2 \leq S$ by rule sub_trans.

Case: $R = R_1 \& R_2$ with S a conjunct of R_2. Symmetric to the previous case.

□

Lemma 2.5.3 (Basic conjuncts inclusion)
If each basic conjunct of R is also a conjunct of S then $S \leq R$.

Proof: By induction on R.

Case: $R = \top^A$. Then $S \leq \top^A$ by rule sub_Tright.

Case: $R = r$ or $R = R_1 \to R_2$.

Then R is a basic conjunct of R (by definition),
so R is a conjunct of S (by assumption),
thus $S \leq R$ (by the previous lemma).

Case: $R = R_1 \& R_2$.

Then $S \leq R_1$ and $S \leq R_2$ by the induction hypothesis,
so $S \leq R_1 \& R_2$ by rule sub_&right.

□

Lemma 2.5.4 (Associativity, commutativity and idempotence of &)
$R \& S \cong S \& R$ and $R_1 \& (R_2 \& R_3) \cong (R_1 \& R_2) \& R_3$ and $R \& R \cong R$

Proof: For each, the basic conjuncts of the left and right hand sides are the same, hence subsorting in each direction follows from the previous lemma. □

This lemma allows us to unambiguously treat composite intersections as intersections of finite sets of sorts. Thus, the following definition is unambiguous up to sort equivalence even though a particular finite set of sorts can be written in many different orders.

Definition 2.5.5
If $\{R_1, \ldots, R_n\}$ is a finite set of sorts each refining type A then $\&\{R_1, \ldots R_n\}$ is defined to be $R_1 \& \ldots \& R_n$, or $\top^A$ if $n = 0$.

The finiteness of refinements modulo equivalence allows us to show that they form a lattice. Finiteness is required here in order to intersect upper bounds.

Theorem 2.5.6 *For each type A, the equivalence classes of refinements of A form a finite lattice.*

Proof: Refinements $R_1, R_2 \sqsubset A$ have a greatest lower bound $R_1 \,\&\, R_2$: it is a lower bound by rules sub_&left1 and sub_&left2, and it is greater than all other lower bounds by sub_&right. The least upper bound $R_1 \vee R_2$ of R_1 and R_2 is the intersection of all upper bounds. More precisely, it is the intersection of a finite set containing a canonical representative $rep(S)$ of each equivalence class of upper bounds, as follows.

$$R_1 \vee R_2 = \&\{rep(S) : R_1 \leq S \text{ and } R_2 \leq S\}.$$

□

2.6 Relating base refinements and finite lattices

We now focus on the structure of the refinements of a base type. By the previous section, for any signature Σ with $\vdash_\Sigma a$:type the equivalence classes of refinements of a will form some finite lattice. We will now additionally show that *every* finite lattice L can be represented by some signature $\Sigma(L)$.

Suppose L has elements $l_1, \ldots, l_n$, with partial order $\preceq$, greatest lower bound operation $\wedge$ and least upper bound operation $\vee$. Then the signature $\Sigma(L)$ contains a single type a, and a refinement $r_i \sqsubset a$ for each lattice element l_i of L. To accurately encode the structure of the lattice, we include subsorting declarations that equate each intersection with the refinement for the greatest lower bound in the lattice. More precisely we include the following declarations in $\Sigma(L)$.

$$\begin{array}{rll} a\text{:type} & & \\ r_i \sqsubset a & & \text{for each } l_i \in L \text{ (choosing distinct } r_i) \\ \top \overset{a}{\leq} r_t & & \text{where } l_t \text{ is the top element of } L \\ r_i \,\&\, r_j \overset{a}{\leq} r_k & \Big\} & \text{for each } l_i, l_j \in L \text{ with } l_i \wedge l_j = l_k. \\ r_k \overset{a}{\leq} r_i \,\&\, r_j & \Big\} & \end{array}$$

We now show that every refinement $R \sqsubset a$ corresponds to some lattice element l_i (the converse is true by construction).

Lemma 2.6.1 (Lattice correspondence)
If $\vdash_{\Sigma(L)} R \sqsubset a$ then there exists $l_i \in L$ such that $\vdash_{\Sigma(L)} R \cong r_i$.

Proof: By induction on the structure of R.

Case: $R = r_i$.

Then $l_i \in L$ and $r_i \cong r_i$.

Case: $R = R_1 \,\&\, R_2$.

$l_i \in L$ with $r_i \cong R_1$ and $l_j \in L$ with $r_j \cong R_2$	Ind. hyp.
$r_i \,\&\, r_j \leq R_i$	Rule sub_&left
$r_i \,\&\, r_j \leq R_j$	Rule sub_&left
$r_i \,\&\, r_j \leq R_i \,\&\, R_j$	Rule sub_&right
$R_i \,\&\, R_j \leq r_i \,\&\, r_j$	Similarly
$r_i \,\&\, r_j \overset{a}{\leq} r_k$ and $r_k \overset{a}{\leq} r_i \,\&\, r_j$ in $\Sigma(L)$ where $l_i \wedge l_j = l_k$	Def. $\Sigma(L)$
$R_1 \,\&\, R_2 \leq r_k$ and $r_k \leq R_1 \,\&\, R_2$	Rule sub_trans, via sub_base
$r_k \cong R_1 \,\&\, R_2$	Def. $\cong$

Case: $R = \top$

$\top \overset{a}{\leq} r_t$ in $\Sigma(L)$ where l_t is the top element of L	Def. $\Sigma(L)$
$\top \leq r_t$	Rule sub_base
$r_t \leq \top$	Rule sub_Tright
$\top \cong r_t$	Def. $\cong$

□

Next, we show that the partial order $\preceq$ is accurately captured by the refinements of a in $\Sigma(L)$.

Theorem 2.6.2 (Lattice structure preservation)
For all i, j we have $\vdash_{\Sigma(L)} r_i \leq r_j$ if and only if $l_i \preceq l_j$.

Proof:

- From right to left:
  $l_i \preceq l_j$ implies $l_i \wedge l_j = l_i$.
  Thus $\Sigma(L)$ contains $r_i \leq r_i \,\&\, r_j$.
  Then $\vdash_{\Sigma(L)} r_i \leq r_j$ by sub_trans and sub_&left2.

- From left to right:
  By induction on the derivation of $\vdash_{\Sigma(L)} r_i \leq r_j$, generalizing the induction hypothesis to:
  For all $i_1 \ldots i_m$ and $j_1 \ldots j_n$

$$\begin{aligned} \text{if} \quad & r_{i_1} \,\&\, \ldots \,\&\, r_{i_m} \leq r_{j_1} \,\&\, \ldots \,\&\, r_{j_n} \\ \text{then} \quad & l_{i_1} \wedge \ldots \wedge l_{i_m} \preceq l_{j_1} \wedge \ldots \wedge l_{j_n}. \end{aligned}$$

  All cases are completely straightforward.

□

Since the partial order $\preceq$ determines the structure of the lattice (including least upper bounds and greatest lower bounds), this theorem implies that the equivalence classes of refinements of a in $\Sigma(L)$ follow the structure of the lattice L.

Thus, every finite lattice can be represented by a signature, and every signature represents a finite lattice. The work of Freeman [Fre94] similarly allows each base type to have any

finite lattice of refinements. However, Freeman directly assumes that the refinements form a lattice, while we isolate our assumptions in a signature, using an extension of the approach of Pfenning [Pfe93].

We do this because it seems more elegant than the explicit global assumptions required in Freeman's presentation. In particular, the declarations provide a convenient way to specify lattices when instantiating our framework, and there is no requirement that the properties of a lattice be verified in each instance. Further, in our case the annotations used by our sort checker for Standard ML require some method for specifying lattices of refinements of opaque types in signatures. The method we use is based directly on the declarations considered in this chapter (see Section 7.7.2). Additionally, for efficiency our sort checker transforms such declarations into a finite lattice, thus making use of the above equivalence between declarations and finite lattices (see Section 8.8).

Our signatures extend those of Pfenning by allowing declarations of the form $R \overset{a}{\leq} S$ instead of only the form $r \overset{a}{\leq} s$. This extension is necessary to allow all finite lattices to be declared. E.g., the following signature has no counterpart in the system presented by Pfenning [Pfe93].

$$a{:}\mathsf{type},\ c{:}a,\ r_1 \sqsubset a,\ r_2 \sqsubset a,\ r_3 \sqsubset a,\ (r_2 \,\&\, r_3 \overset{a}{\leq} r_1).$$

2.7 Algorithmic base subsorting

For efficiency, our implementation of sort checking represents base refinements as elements of finite lattices, with each element corresponding to an equivalence class. This will be reflected in the presentation in Chapter 3. In this chapter we have a more abstract presentation, which requires a different technique for determining inclusion between base refinements. This section presents one such algorithm.

This algorithm works by determining all upper bounds s for a particular base refinement R, which is done as follows.

- Each r in R is an upper bound for R.
- When $S_1 \overset{a}{\leq} S_2$ is in Σ, if each s_1 in S_1 is an upper bound for R then each s_2 in S_2 is an upper bound for R.

The following judgment presents this algorithm more formally. It requires $R \sqsubset a$ and $s \sqsubset a$ in order to be well formed.

$\vdash_\Sigma R \overset{a}{\trianglelefteq} s$ $\quad R$ is determined to be a base subsort of s.

$$\frac{s \text{ in } R}{R \overset{a}{\trianglelefteq} s} \qquad \frac{r_1 \& \ldots \& r_n \overset{a}{\leq} S \text{ in } \Sigma \qquad R \overset{a}{\trianglelefteq} r_1 \ \ \ldots \ \ R \overset{a}{\trianglelefteq} r_n \qquad s \text{ in } S}{R \overset{a}{\trianglelefteq} s}$$

Unlike the other algorithms in this dissertation, the sense in which these rules describe an algorithm is different from the standard interpretation as a backward chaining logic program. If we take this interpretation, the above algorithm may non-terminate (e.g., when Σ contains $s \overset{a}{\leq} s$).

Instead, the intended algorithmic interpretation of these rules is as a tabling, forward chaining logic program: given a particular R, the rules are applied to iteratively increase the set of s for which $R \stackrel{a}{\trianglelefteq} s$ is known. When no rule will add a new s to the set, the algorithm has competed. The termination of the algorithm is straightforward with this interpretation: each step of the algorithm must add $R \stackrel{a}{\trianglelefteq} s$ for some new s, and there are only finitely many s. As observed by Ganzinger and McAllester [GM02], this style of logic programming allows many efficient algorithms to be elegantly expressed as inference rules which can not easily be expressed with a backward chaining interpretation.

This interpretation only affects the way we reason about the termination of the algorithm. It does not affect the way we prove soundness and completeness for the algorithm: we still use the standard techniques of induction over the structure of derivations. Soundness can be proved relatively easily, as follows.

Theorem 2.7.1 (Soundness of $\stackrel{a}{\trianglelefteq}$)
If $R \stackrel{a}{\trianglelefteq} s$ then $R \leq s$.

Proof: By induction on the structure of $R \stackrel{a}{\trianglelefteq} s$.

Case: $\dfrac{s \text{ in } R}{R \stackrel{a}{\trianglelefteq} s}$

$R \leq s$ — By Lemma 2.5.2 (Conjunct inclusion)

Case: $\dfrac{r_1 \& \ldots \& r_n \stackrel{a}{\leq} S \text{ in } \Sigma \qquad R \stackrel{a}{\trianglelefteq} r_1 \quad \ldots \quad R \stackrel{a}{\leq} r_n \qquad s \text{ in } S}{R \stackrel{a}{\trianglelefteq} s}$

$R \leq r_i$ for each r_i — Ind. hyp.
$R \leq r_1 \& \ldots \& r_n$ — Rule sub_&right, repeatedly
$r_1 \& \ldots \& r_n \leq S$ — Rule sub_base
$R \leq S$ — Rule sub_trans
$S \leq s$ — Lemma 2.5.2 (Conjunct inclusion)
$R \leq s$ — Rule sub_trans

□

The proof of completeness of $\stackrel{a}{\trianglelefteq}$ requires the following lemma.

Lemma 2.7.2 (Transitivity of $\stackrel{a}{\trianglelefteq}$)
If $R_1 \stackrel{a}{\trianglelefteq} s_2$ for each s_2 in R_2
and $R_2 \stackrel{a}{\trianglelefteq} s_3$
then $R_1 \stackrel{a}{\trianglelefteq} s_3$.

Proof: By induction on the structure of the derivation of $R_2 \stackrel{a}{\trianglelefteq} s_3$.

Case: $\dfrac{s_3 \text{ in } R_2}{R_2 \stackrel{a}{\trianglelefteq} s_3}$

$R_1 \overset{a}{\trianglelefteq} s_2$ for each s_2 in R_2 — Assumption

$R_1 \overset{a}{\trianglelefteq} s_3$ — Instantiating

Case: $\dfrac{r_1 \& \ldots \& r_n \overset{a}{\leq} S \text{ in } \Sigma \qquad R_2 \overset{a}{\trianglelefteq} r_1 \quad \ldots \quad R_2 \overset{a}{\leq} r_n \qquad s_3 \text{ in } S}{R_2 \overset{a}{\trianglelefteq} s_3}$

$R_1 \overset{a}{\trianglelefteq} s_2$ for each s_2 in R_2 — Assumption

$R_1 \overset{a}{\trianglelefteq} r_i$ for each r_i — Ind. hyp. (for each)

$R_1 \overset{a}{\trianglelefteq} s_3$ — Rule for $R \overset{a}{\leq} S$ in Σ

□

Theorem 2.7.3 (Completeness of $\overset{a}{\trianglelefteq}$)
If $R \leq S$ and s in S then $R \overset{a}{\trianglelefteq} s$.

Proof: By induction on the structure of the derivation of $R \leq S$.

Case: $\dfrac{R \overset{a}{\leq} S \text{ in } \Sigma}{R \leq S}\,\mathsf{sub\_base}$

$R \overset{a}{\trianglelefteq} r$ for each r in R — Rule for r in R

s in S — Assumption

$R \overset{a}{\trianglelefteq} s$ — Rule for $R \overset{a}{\leq} S$

Case: $\dfrac{}{R \leq R}\,\mathsf{sub\_reflex}$

s in R — Assumption

$R \overset{a}{\trianglelefteq} s$ — Rule for s in R

Case: $\dfrac{R \leq R_2 \quad R_2 \leq S}{R \leq S}\,\mathsf{sub\_trans}$

$R \overset{a}{\trianglelefteq} s_2$ for each s_2 in R_2 — Ind. hyp.

$R_2 \overset{a}{\trianglelefteq} s$ — Ind. hyp.

$R \overset{a}{\trianglelefteq} s$ — Lemma 2.7.2 (Transitivity of $\overset{a}{\trianglelefteq}$)

Case: $\dfrac{}{S \,\&\, R_2 \leq S}\,\mathsf{sub\_\&left1}$

s in S — Assumption

s in $S \,\&\, R_2$ — Def. "in"

$S \,\&\, R_2 \overset{a}{\trianglelefteq} s$ — Rule for r in R

Case: $\dfrac{}{R_1 \,\&\, S \leq S}\,\mathsf{sub\_\&left2}$

Symmetric to the previous case.

Case: $\dfrac{R \leq S_1 \qquad R \leq S_2}{R \leq S_1 \,\&\, S_2}\,\mathsf{sub\_\&right}$

s in $S_1 \,\&\, S_2$ — Assumption

s in S_1 or s in S_2 — Def. "in"

$R \overset{a}{\trianglelefteq} s$ — Ind. hyp. in each case

Case: $\dfrac{}{R \leq \top^A}\,\mathsf{sub\_Tright}$

s in $\top^A$ cannot occur.

□

The following theorem captures the sense in which the judgment $R \overset{a}{\trianglelefteq} s$ can be used to determine whether the subsorting $R \leq S$ holds for any R and S, i.e. not only when S is a base sort.

Theorem 2.7.4 (Correctness of $\overset{a}{\trianglelefteq}$)
$R \overset{a}{\trianglelefteq} s$ for each s in S if and only if $R \leq S$.

Proof:

Left to right

$R \leq s$ for each s in S — Lemma 2.7.1 (Soundness of $\overset{a}{\trianglelefteq}$)

$R \leq S$ — Rule sub_&right, repeatedly

Right to left

$R \leq s$ for each s in S — Lemma 2.5.2 (Conjunct inclusion)

$R \overset{a}{\trianglelefteq} s$ for each s in S — Lemma 2.7.3 (Completeness of $\overset{a}{\trianglelefteq}$)

□

2.8 Declarative sort assignment

The sort assignment judgment has the following form, where we require $\Gamma \vdash M : A$, $\Delta \sqsubset \Gamma$ and $R \sqsubset A$ in order for the judgment to be well formed.

$\Delta \vdash M \in R$ Term M has sort R in context Δ.

The sort assignment rules are very similar to those for a system with general intersection types (such as in [Rey91]).

$$\frac{x{\in}R \text{ in } \Delta}{\Delta \vdash x \in R}\,\mathsf{sa\_var} \qquad \frac{c{\in}R \text{ in } \Sigma}{\Delta \vdash c \in R}\,\mathsf{sa\_const}$$

$$\frac{\Delta, x{\in}R \vdash M \in S}{\Delta \vdash \lambda x{:}A.M \in R \to S}\,\mathsf{sa\_lam} \qquad \frac{\Delta \vdash M \in R \to S \quad \Delta \vdash N \in R}{\Delta \vdash M\,N \in S}\,\mathsf{sa\_app}$$

$$\frac{\Delta \vdash M \in R \quad \Delta \vdash M \in S}{\Delta \vdash M \in R \& S}\,\mathsf{sa\_inter} \qquad \frac{}{\Delta \vdash M \in \top^A}\,\mathsf{sa\_top}$$

$$\frac{\Delta \vdash M \in R \quad R \leq S}{\Delta \vdash M \in S}\,\mathsf{sa\_subs}$$

Here each abstraction includes the type A of the variable, so the well formedness conditions restrict the choice of the sort of the variable to refinements of A. This could be made explicit in the rule, but we have chosen not to because we feel it is a natural consequence of our approach to using well formedness conditions to relate sorts and types.

Our calculus satisfies the usual substitution lemmas with respect to sorts, subject reduction and sort preservation theorems with respect to β-reduction. We omit proofs, since the proofs for general intersection types (see e.g. [CDV81]) can easily be adapted.

We also have the following simple theorem, which can be proved more easily in our situation than with general intersection types: we can take advantage of the finiteness of the equivalence classes of refinements of each type.

Theorem 2.8.1 (Principal sorts)
If $\Delta \vdash M : A$ then there exists $R \sqsubset A$ such that $\Delta \vdash M \in R$ and every S such that $\Delta \vdash M \in S$ satisfies $R \leq S$.

Proof: Choose R to be the intersection of a finite set of sorts containing one representative R_i of each equivalence class that satisfies $\Delta \vdash M \in R_i$. □

2.9 Algorithmic subsorting

The declarative sorting and subsorting rules are quite intuitive, but they do not specify a strategy for checking a term. We first treat the issue of constructing an algorithm for determining subsorting. We address the more difficult issue of algorithmic sort checking in Section 2.10. Our subsorting algorithm is similar to that used by Freeman [Fre94].

A quite different algorithm has been designed by Reynolds [Rey91] for the programming language Forsythe, and similar algorithms have been considered by Pierce [Pie91, Pie97]. Interestingly, Reynolds' algorithm cannot be easily extended to our base subsorting relations. For example, suppose we have the following signature (from Section 2.6).

$$\Sigma \;=\; a{:}\mathsf{type},\ c{:}a,\ r_1{\sqsubset}a,\ r_2{\sqsubset}a,\ r_3{\sqsubset}a,\ (r_2 \& r_3 \overset{a}{\leq} r_1)$$

Then we can form the following subsorting derivation.

$$\dfrac{\dfrac{}{(r_1 \to r_2)\,\&\,(r_1 \to r_3) \;\le\; r_1 \to (r_2\,\&\,r_3)} \qquad \dfrac{r_2\,\&\,r_3 \overset{a}{\le} r_1 \text{ in } \Sigma}{r_2\,\&\,r_3 \le r_1}}{(r_1 \to r_2)\,\&\,(r_1 \to r_3) \;\le\; r_1 \to r_1.}$$

However, an algorithm based on Reynolds' approach would incorrectly determine that this subsorting inclusion is false. Roughly, this is because the algorithm depends upon the occurrence of an intersection within the sort on the right to determine potential uses of the distributivity rule. But when we have a declaration like $r_2 \& r_3 \overset{a}{\le} r_1$ there are some subsorting instances which require the distributivity rule but have no such occurrence.

The subsorting algorithm we present here is based on a variant of that presented by Freeman. More precisely, it is based on the "more efficient" variant mentioned on page 119 of [Fre94]. Freeman focuses on a "less efficient" algorithm because it is easier to prove correct, and fits better with the representations of sorts used in his sort inference algorithm. However, the less efficient algorithm seems less likely to scale to complex and higher-order types because it requires explicit enumeration of the refinements of a type.

We describe our subsorting algorithm using a judgment $R \trianglelefteq S$ with algorithmic rules. This judgment depends on an ancillary judgment $R \trianglelefteq S_1 \overset{\Rightarrow}{\to} S_2$ which takes R and S_1 as input and synthesizes a minimum S_2 such that $R \le S_1 \to S_2$. This judgment in turn requires the complement $R \not\trianglelefteq S$ of algorithmic subsorting: formally this means that $R \trianglelefteq S$ fails, i.e., that the standard backward-chaining search for a proof of $R \trianglelefteq S$ terminates without finding a proof. The rules are designed so that the algorithm always terminates, which can be shown by induction on the term and the sort R, lexicographically. The complement judgment $R \not\trianglelefteq S$ is only included as a premise to reduce the non-determinism in the algorithm.

Here and in later algorithmic judgments we use the notation "$\Rightarrow$" to emphasize outputs from a judgment. Later we will also use the notation "$\Leftarrow$" to emphasize inputs.

The judgments have the following forms, where the first two require R and S to be compatible in order to be well formed, and the third requires that R be compatible with $S_1 \to S_2$.

$R \trianglelefteq S$	R is determined to be a subsort of S.
$R \not\trianglelefteq S$	The complement of $R \trianglelefteq S$, i.e., $R \trianglelefteq S$ fails.
$R \trianglelefteq S_1 \overset{\Rightarrow}{\to} S_2$	S_2 is synthesized as the minimum sort such that $R \le S_1 \to S_2$, given R and S_1

$$\dfrac{R \overset{a}{\trianglelefteq} s}{R \trianglelefteq s}\ \mathsf{subalg\_base} \qquad \dfrac{R \trianglelefteq S_1 \qquad R \trianglelefteq S_2}{R \trianglelefteq S_1\,\&\,S_2}\ \mathsf{subalg\_inter}$$

$$\dfrac{R \trianglelefteq S_1 \overset{\Rightarrow}{\to} R_2 \qquad R_2 \trianglelefteq S_2}{R \trianglelefteq S_1 \to S_2}\ \mathsf{subalg\_arrow} \qquad \dfrac{}{R \trianglelefteq \top^A}\ \mathsf{subalg\_top}$$

$$\frac{S_1 \trianglelefteq R_1}{R_1 \to R_2 \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} R_2}\ \textsf{apsrt\_arrow} \qquad \frac{S_1 \not\trianglelefteq R_1}{R_1 \to R_2 \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} \top^B}\ \textsf{apsrt\_arrow\_top}$$

$$\frac{R \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} S_2 \qquad R' \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} S_2'}{R \,\&\, R' \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} (S_2 \,\&\, S_2')}\ \textsf{apsrt\_inter} \qquad \frac{}{\top^{A \to B} \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} \top^B}\ \textsf{apsrt\_top}$$

To show that the algorithmic subsorting judgment is equivalent to the declarative subsorting judgment, we first need some lemmas.

Lemma 2.9.1 (Algorithmic subsorting for conjuncts)

1. *If S is a conjunct of R then $R \trianglelefteq S$.*
2. *If $S = S_1 \to S_2$ is a conjunct of R then $R \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} R_2$ for some R_2 with S_2 a conjunct of R_2.*

Proof: By induction on the structure of S, R lexicographically.

For part 1:

Case: $S = s$. By Lemma 2.7.3 (Completeness of $\stackrel{a}{\trianglelefteq}$).

Case: $S = S_1 \,\&\, S_2$. Straightforward, applying the ind. hyp. to S_1 and S_2.

Case: $S = \top$. By rule subalg_top.

Case: $S = S_1 \to S_2$.

$R \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} R_2$ with S_2 a conjunct of R_2	Part 2 of ind. hyp.
$R_2 \trianglelefteq S_2$	Part 1 of ind. hyp.
$R \trianglelefteq S_1 \to S_2$	Rule subalg_arrow

For part 2:

Case: $R = R_1 \to R_2$.
Then $R_1 = S_1$ and $R_2 = S_2$ and so R_1 is a conjunct of S_1.
Applying the ind. hyp. yields $S_1 \trianglelefteq R_1$, so $R_1 \to R_2 \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} R_2$.

Case: $R = R_1 \,\&\, R_2$. By applying the ind. hyp. to either R_1 or R_2.

Case: $R = \top$. Impossible: S is a conjunct of R.

□

Corollary 2.9.2 (Algorithmic subsorting reflexivity)
For every well-formed sort S we have $S \trianglelefteq S$.

Lemma 2.9.3 (Properties of $\trianglelefteq^{\stackrel{\Rightarrow}{\to}}$) *For every $R \sqsubset A_1 \to A_2$ and $S_1 \sqsubset A_1$ there is a unique S_2 such that $R \trianglelefteq S_1 \stackrel{\Rightarrow}{\to} S_2$. Further, the size of S_2 is smaller than or equal to the size of R.*

Proof: By induction on R.

Case: $R = R_1 \,\&\, R_2$. Straightforward, applying the ind. hyp. to R_1 and R_2.

Case: $R = \top^{A \to B}$. By rule apsrt_top.

Case: $R = R_1 \to R_2$.

Then exactly one of apsrt_arrow and apsrt_arrow_top applies,
each is straightforward.

□

Lemma 2.9.4 (Algorithmic subsorting for intersections)

If $R \trianglelefteq S$ then $R \,\&\, R' \trianglelefteq S$ and $R' \,\&\, R \trianglelefteq S$.

Proof: By induction on S.

Case: $S = s$.

From the corresponding property of $\leq$, via Lemmas 2.7.1 and 2.7.3
(Soundness and Completeness of $\overset{a}{\trianglelefteq}$).

Case: $S = S_1 \,\&\, S_2$. Straightforward, applying the ind. hyp. to S_1 and S_2.

Case: $S = \top$. By rule subalg_top.

Case: $S = S_1 \to S_2$.

$R \trianglelefteq S_1 \overset{\Rightarrow}{\to} R_2$ and $R_2 \trianglelefteq S_2$	Inversion.
$R' \trianglelefteq S_1 \overset{\Rightarrow}{\to} R_2'$ for some R_2'	Properties of $\trianglelefteq^{\overset{\Rightarrow}{\to}}$ (2.9.3)
$R \,\&\, R' \trianglelefteq S_1 \overset{\Rightarrow}{\to} (R_2 \,\&\, R_2')$	Rule apsrt_inter
$R_2 \,\&\, R_2' \trianglelefteq S_2$	Ind. hyp.
$R \,\&\, R' \trianglelefteq S_1 \to S_2$	Rule subalg_arrow

□

The main lemma required for the correctness of our algorithm is one that validates transitivity. Its proof requires generalizing the induction hypothesis to include two additional results which are forms of transitivity involving the judgment $R \trianglelefteq S_1 \to S_2$. We use induction over the sum of the sizes of the sorts: a simple induction on the structure of the two derivations is insufficient, because we sometimes need to apply part of the induction hypothesis to a derivation constructed via an earlier application of the induction hypothesis. Further, the two derivations are swapped for some appeals to the induction hypothesis, due to the contravariance of $\to$. (It appears that this proof would also work by induction first on the type A refined by R followed by R, R_1 and R_2.)

Lemma 2.9.5 (Algorithmic subsorting transitivity)

1. *If $R_1 \trianglelefteq R$ and $R \trianglelefteq R_2$ then $R_1 \trianglelefteq R_2$.*
2. *If $R_1 \trianglelefteq R$ and $R \trianglelefteq R_2 \overset{\Rightarrow}{\to} S$ then $R_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S_1$ for some $S_1 \trianglelefteq S$.*
3. *If $R_1 \trianglelefteq R \overset{\Rightarrow}{\to} S$ and $R_2 \trianglelefteq R$ then $R_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S_2$ for some $S_2 \trianglelefteq S$.*

Proof:
By induction on the sum of the sizes of R_1, R_2 and R, making use of the above lemmas.

For the first part, we treat each possible derivation for $R \trianglelefteq R_2$.

Case: $R_1 \trianglelefteq R$ and $\dfrac{R \overset{a}{\trianglelefteq} s_2}{R \trianglelefteq s_2}$.

Using Lemma 2.7.2 (Transitivity of $\overset{a}{\trianglelefteq}$),
via a small subinduction to show $R_1 \overset{a}{\trianglelefteq} s$ for each s in R.

Case: $R_1 \trianglelefteq R$ and $\dfrac{R \trianglelefteq R_{21} \qquad R \trianglelefteq R_{22}}{R \trianglelefteq R_{21} \,\&\, R_{22}}$

Via the ind. hyp. on R_{21} and R_{22}.

Case: $R_1 \trianglelefteq R$ and $\dfrac{}{R \trianglelefteq \top}$

By rule subalg_top.

Case: $R_1 \trianglelefteq R$ and $\dfrac{R \trianglelefteq R_{21} \overset{\Rightarrow}{\to} R'_{22} \qquad R'_{22} \trianglelefteq R_{22}}{R \trianglelefteq R_{21} \to R_{22}}$.

$R_1 \trianglelefteq R_{21} \overset{\Rightarrow}{\to} R''_{22}$ for some $R''_{22} \trianglelefteq R'_{22}$	Part 2 of ind. hyp.
R''_{22} smaller than R_1 and R'_{22} smaller than R	Lemma 2.9.3
$R''_{22} \trianglelefteq R_{22}$	Part 1 of ind. hyp.
$R_1 \trianglelefteq R_{21} \to R_{22}$	Rule subalg_arrow

For the second part, we have the following cases.

Case: $\dfrac{R_1 \trianglelefteq R'_1 \qquad R_1 \trianglelefteq R'_2}{R_1 \trianglelefteq R'_1 \,\&\, R'_2}$ and $\dfrac{R'_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S'_1 \qquad R'_2 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S'_2}{R'_1 \,\&\, R'_2 \trianglelefteq R_2 \overset{\Rightarrow}{\to} (S'_1 \,\&\, S'_2)}$

$R_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S''_1$ for some $S''_1 \trianglelefteq S'_1$	Ind. hyp.
$R_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S''_2$ for some $S''_2 \trianglelefteq S'_2$	Ind. hyp.
$S''_1 = S''_2$	Lemma 2.9.3
$S''_1 \trianglelefteq S'_1 \,\&\, S'_2$	Rule subalg_inter

Case: $\dfrac{}{R_1 \trianglelefteq \top}$ and $\dfrac{}{\top \trianglelefteq R_2 \overset{\Rightarrow}{\to} \top}$

$R_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} S'$ for some S'	Lemma 2.9.3
$S' \trianglelefteq \top$	Rule subalg_top

Case: $\dfrac{R_1 \trianglelefteq R'_1 \overset{\Rightarrow}{\to} R''_2 \qquad R''_2 \trianglelefteq R'_2}{R_1 \trianglelefteq R'_1 \to R'_2}$ and $\dfrac{R_2 \trianglelefteq R'_1}{R'_1 \to R'_2 \trianglelefteq R_2 \overset{\Rightarrow}{\to} R'_2}$

$R_1 \trianglelefteq R_2 \overset{\Rightarrow}{\to} R'''_2$ for some $R'''_2 \trianglelefteq R''_2$	Ind. hyp (part 3)
$R'''_2 \trianglelefteq R'_2$	Ind. hyp. (part 1)

Case: $\dfrac{R_1 \trianglelefteq R_1' \stackrel{\Rightarrow}{\rightarrow} R_2'' \qquad R_2'' \trianglelefteq R_2'}{R_1 \trianglelefteq R_1' \rightarrow R_2'}$ and $\dfrac{R_2 \not\trianglelefteq R_1'}{R_1' \rightarrow R_2' \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} \top}$

$R_1 \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} R_2'''$ for some R_2'''	Lemma 2.9.3
$R_2''' \trianglelefteq \top$	Rule subalg_top

For the third part, we consider each possible derivation for $R_1 \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} S$.

Case: $\dfrac{R_{11} \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} S_1 \qquad R_{12} \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} S_2}{R_{11} \,\&\, R_{12} \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} (S_1 \,\&\, S_2)}$ and $R_2 \trianglelefteq R$

$R_{11} \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} S_1'$ with $S_1' \trianglelefteq S_1$	Ind. hyp.
$R_{12} \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} S_2'$ with $S_2' \trianglelefteq S_2$	Ind. hyp.
$R_{11} \,\&\, R_{12} \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} S_1' \,\&\, S_2'$	Rule apsrt_inter
$S_1' \,\&\, S_2' \trianglelefteq S_1$	Lemma 2.9.4
$S_1' \,\&\, S_2' \trianglelefteq S_2$	Lemma 2.9.4
$S_1' \,\&\, S_2' \trianglelefteq S_1 \,\&\, S_2$	Rule subalg_inter

Case: $\dfrac{}{\top \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} \top}$ and $R_2 \trianglelefteq R$

$\top \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} \top$	Rule apsrt_top
$\top \trianglelefteq \top$	Rule subalg_top

Case: $\dfrac{R \trianglelefteq R_{11}}{R_{11} \rightarrow R_{12} \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} R_{12}}$ and $R_2 \trianglelefteq R$

$R_2 \trianglelefteq R_{11}$	Ind. hyp. (part 1)
$R_{11} \rightarrow R_{12} \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} R_{12}$	Rule apsrt_arrow
$R_{12} \trianglelefteq R_{12}$	Reflexivity of $\trianglelefteq$ (2.9.2)

Case: $\dfrac{R \not\trianglelefteq R_{11}}{R_{11} \rightarrow R_{12} \trianglelefteq R \stackrel{\Rightarrow}{\rightarrow} \top}$ and $R_2 \trianglelefteq R$

$R_{11} \rightarrow R_{12} \trianglelefteq R_2 \stackrel{\Rightarrow}{\rightarrow} S_2$ for some S_2	Lemma 2.9.3
$S_2 \trianglelefteq \top$	Rule subalg_top

□

We are now in a position to prove correctness of our subsorting algorithm.

Theorem 2.9.6 (Algorithmic subsorting correctness)
$R \trianglelefteq S$ if and only if $R \leq S$.

Proof:
Left to right: by a straightforward induction on the derivation of $R \trianglelefteq S$, strengthening the

induction hypothesis to also include: if $R \trianglelefteq S_1 \stackrel{\Rightarrow}{\rightarrow} S_2$ then $R \leq S_1 \rightarrow S_2$.
There is one case for each algorithmic rule, and each essentially shows that the rule corresponds to a derived rule in the declarative system.

Right to left: by induction on the derivation of $R \leq S$, making use of the above lemmas.

- The case for sub_reflex uses Reflexivity of $\trianglelefteq$ (2.9.2).
- The case for sub_trans uses Transitivity of $\trianglelefteq$ (2.9.5).
- The case for sub_base uses Correctness of $\stackrel{a}{\trianglelefteq}$ (2.7.4).
- The cases for sub_&left1 and sub_&left2 use Lemma 2.9.1.
- The case for sub_&right uses rule subalg_inter.
- The case for sub_Tright uses rule subalg_top.
- The case for sub_arrow uses rule apsrt_arrow followed by subalg_arrow.
- The case for sub_&dist uses rule apsrt_arrow twice, followed by apsrt_inter, followed by subalg_arrow (via Reflexivity of $\trianglelefteq$).
- The case for sub_Tdist uses rule apsrt_top followed by subalg_arrow (via Reflexivity of $\trianglelefteq$).

□

2.10 Bidirectional sort checking

2.10.1 Syntax for annotations

Our bidirectional algorithm allows the programmer to specify sorts using annotations, resulting in an efficient approach to sort checking. We thus extend the language of terms with a construct for terms annotated with a list of sorts that may be assigned to the term: $M \in R_1, \ldots, R_n$. This is similar to the type annotations with alternatives in Forsythe [Rey96], except that in Forsythe the annotations are placed on variable bindings rather than on terms.

$$\begin{array}{rrcl} \text{Terms} & M, N & ::= & \ldots \mid (M \in L) \\ \text{Sort Constraints} & L & ::= & \cdot \mid L, R \end{array}$$

We extend the judgments for validity of terms and declarative sort assignment to this construct, as follows.

$$\frac{R_1 \sqsubset A \quad \ldots \quad R_n \sqsubset A \qquad \Gamma \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma (M \in R_1, \ldots, R_n) : A}\ \textsf{objann} \qquad \frac{R \text{ in } L \qquad \Delta \vdash M \in R}{\Delta \vdash (M \in L) \in R}\ \textsf{sa\_ann}$$

The rule for validity of terms objann checks that the annotations on a term refine the type of the term. The rule for sort assignment sa_ann restricts the sorts which may be assigned to the annotated term. This is necessary in order to obtain a correspondence with the sorts

used during sort checking. Including this rule in the declarative system allows a programmer to reason about the sorts associated with their annotated program without having to reason in terms of the sort checking algorithm. This point will be discussed in greater detail in the next chapter (in Section 3.11.3).

Annotations are only used during sort-checking, so there is no reason to extend notions such as reduction to annotations. Instead we define the function $\|\cdot\|$ that erases annotations from a term, reflecting our intention that sort annotations should be removed after sort checking.

$$\begin{array}{rcl} \|M{\in}L\| & = & \|M\| \\ \|x\| & = & x \\ \|c\| & = & c \\ \|\lambda x{:}A.M\| & = & \lambda x{:}A.\|M\| \\ \|M\ N\| & = & \|M\|\ \|N\| \end{array}$$

We now demonstrate that the function $\|\cdot\|$ preserves types and sorts, justifying the notion that annotations can be removed after sort checking.

Theorem 2.10.1 (Typing Erasure)
If $\Gamma \vdash_\Sigma M : A$ *then* $\Gamma \vdash_\Sigma \|M\| : A$.

Proof: By a straightforward induction on the structure of the derivation. We show the case for the rule objann. The remaining cases simply rebuild the derivation by mirroring the structure of the given derivation.

Case:
$$\frac{R_1 \sqsubset A \quad \ldots \quad R_n \sqsubset A \qquad \overset{\mathcal{E}}{\Gamma \vdash_\Sigma N : A}}{\Gamma \vdash_\Sigma (N{\in}L) : A}\ \mathsf{objann}$$

Then $\|M\| = \|N{\in}L\| = \|N\|$ and we apply the induction hypothesis to $\mathcal{E}$ to obtain $\Gamma \vdash_\Sigma \|N\| : A$, as required.

□

Theorem 2.10.2 (Sorting Erasure)
If $\Delta \vdash M \in R$ *then* $\Delta \vdash \|M\| \in R$.

Proof: By a straightforward induction on the structure of the derivation. We show the cases for rules sa_ann and sa_inter. The remaining cases simply rebuild the derivation by mirroring the structure of the given derivation.

Case:
$$\frac{R \text{ in } L \qquad \overset{\mathcal{D}}{\Delta \vdash N \in R}}{\Delta \vdash (N{\in}L) \in R}\ \mathsf{sa\_ann}$$

Then $\|M\| = \|N{\in}L\| = \|N\|$ and we apply the induction hypothesis to $\mathcal{D}$ to obtain $\Delta \vdash \|N\| \in R$, as required.

Case:
$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash M \in R_1 & \Delta \vdash M \in R_2\end{array}}{\Delta \vdash M \in R_1 \,\&\, R_2}\ \mathsf{sa\_inter}$$

Then, we apply the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ to obtain $\Delta \vdash \|M\| \in R_1$ and $\Delta \vdash \|M\| \in R_2$. We then apply rule sa_inter to rebuild the derivation.

□

2.10.2 Algorithm

The bidirectional checking algorithm uses two subclasses of terms, *inferable terms* and *checkable terms*. We also distinguish the sorts R^{n} that do not have an intersection at the top level, in order to make our rules deterministic.

$$\begin{array}{rrl}\text{Inferable Terms} & I & ::= c \mid x \mid I\,C \mid (C{\in}L) \\ \text{Checkable Terms} & C & ::= I \mid \lambda x{:}A.C \\ \text{Non-intersection Sort} & R^{\mathsf{n}} & ::= r \mid R \to S\end{array}$$

We note that the checkable terms without annotations are exactly the normal forms. Thus, annotations are only required at the top level, and where there is a redex in a term.

The bidirectional sort checking algorithm is presented as two judgments with algorithmic rules: one that checks a checkable term against a sort, and one that infers a principal sort for an inferable term. We also require the complement of the checking judgment, and an ancillary judgment that projects the sort of a function onto an argument to synthesize the principal sort for an application. In each of these judgments the context is always an input: the sorts of all variables in scope are always known during an execution of the algorithm. The algorithm always terminates: each premise has a smaller term than the conclusion, or the same term and a smaller sort R, or the premise involves an earlier judgment than the conclusion (according the order below).

We write the well-formedness conditions at the end of the description of each judgment (prefixed by "where").

$\Delta \vdash I \overset{\Rightarrow}{\in} R$	R is synthesized as a principal sort for I under Δ, where $\Delta \sqsubset \Gamma$, $R \sqsubset A$, $\Gamma \vdash C : A$.
$\Delta \vdash C \overset{\Leftarrow}{\in} R$	Term C checks against sort R under Δ, where $\Delta \sqsubset \Gamma$, $R \sqsubset A$, $\Gamma \vdash C : A$.
$\Delta \vdash C \overset{\Leftarrow}{\notin} R$	The complement of $\Delta \vdash C \overset{\Leftarrow}{\in} R$, i.e., $\Delta \vdash C \overset{\Leftarrow}{\in} R$ fails. where $\Delta \sqsubset \Gamma$, $R \sqsubset A$, $\Gamma \vdash C : A$.
$\Delta \vdash R \trianglelefteq C \overset{\Rightarrow}{=} S$	S is synthesized as a principal sort for an application of a function with sort R to C, where $\Delta \sqsubset \Gamma$, $R \sqsubset A \to B$, $S \sqsubset B$, $\Gamma \vdash C : A$.

We have one checking rule for each kind of checkable term, plus one each for & and $\top$. The rule for $\lambda x{:}A.C$ adds an appropriate assumption for the variable based on the sort being checked

against, ensuring that the sorts of all variables in the context are known during the algorithm. The rule for an inferable term non-deterministically synthesizes a sort and then compares it with the goal. It is restricted to non-intersection sorts R^{n} to avoid unnecessary non-determinism. The checking rules for & and $\top$ mirror those in the declarative system.

We have one synthesis rule for each kind of inferable term. The rules for variables and constants are straightforward. The rule for applications synthesizes a sort for the function, and then uses the auxiliary judgment to "apply" this sort over the argument. The rule si_ann for annotations synthesizes a sort for $(C{\in}\{R_1, \ldots, R_m, S_1, \ldots, S_n\})$. Here $\{R_1, \ldots, R_m, S_1, \ldots, S_n\}$ indicates that we are treating the annotation like a set: in particular the sorts $S_1, \ldots S_n$ need not be those at the end of the annotation. The rule checks C against each sort in the annotation, and we partition the sorts into $R_1, \ldots, R_m$ for which the check fails, and $S_1, \ldots, S_n$ for which it succeeds. We then synthesize the principal sort by intersecting the latter sorts.

The rules for the auxiliary judgment try each conjunct $R_1 \to R_2$ of the function sort, intersecting those R_2 for which C checks against the corresponding R_1 (and using $\top$ otherwise).

$$\frac{\Delta \vdash C \overset{\Leftarrow}{\in} R \qquad \Delta \vdash C \overset{\Leftarrow}{\in} S}{\Delta \vdash C \overset{\Leftarrow}{\in} R \,\&\, S}\ \mathsf{sc\_inter} \qquad \frac{}{\Delta \vdash C \overset{\Leftarrow}{\in} \top^A}\ \mathsf{sc\_top}$$

$$\frac{\Delta, x{\in}R \vdash C \overset{\Leftarrow}{\in} S}{\Delta \vdash \lambda x{:}A.C \overset{\Leftarrow}{\in} R \to S}\ \mathsf{sc\_lam} \qquad \frac{\Delta \vdash I \overset{\Rightarrow}{\in} S \qquad S \trianglelefteq R^{\mathsf{n}}}{\Delta \vdash I \overset{\Leftarrow}{\in} R^{\mathsf{n}}}\ \mathsf{sc\_atom}$$

. . . . . .

$$\frac{x{\in}R \text{ in } \Delta}{\Delta \vdash x \overset{\Rightarrow}{\in} R}\ \mathsf{si\_var} \qquad \frac{c{\in}R \text{ in } \Sigma}{\Delta \vdash c \overset{\Rightarrow}{\in} R}\ \mathsf{si\_const} \qquad \frac{\Delta \vdash I \overset{\Rightarrow}{\in} R \qquad \Delta \vdash R \trianglelefteq C \overset{\Rightarrow}{\to} S}{\Delta \vdash I\, C \overset{\Rightarrow}{\in} S}\ \mathsf{si\_app}$$

$$\frac{\begin{array}{cccc} \Delta \vdash C \overset{\Leftarrow}{\notin} R_1 & \ldots & \Delta \vdash C \overset{\Leftarrow}{\notin} R_m \\ \Delta \vdash C \overset{\Leftarrow}{\in} S_1 & \ldots & \Delta \vdash C \overset{\Leftarrow}{\in} S_n \end{array}}{\Delta \vdash (C{\in}\{R_1, \ldots, R_m, S_1, \ldots, S_n\}) \overset{\Rightarrow}{\in} \&\{S_1, \ldots, S_n\}}\ \mathsf{si\_ann}$$

. . . . . .

$$\frac{\Delta \vdash C \overset{\Leftarrow}{\in} R_1}{\Delta \vdash R_1 \to R_2 \trianglelefteq C \overset{\Rightarrow}{\to} R_2}\ \mathsf{aptm\_arrow} \qquad \frac{\Delta \vdash C \overset{\Leftarrow}{\notin} R_1}{\Delta \vdash R_1 \to R_2 \trianglelefteq C \overset{\Rightarrow}{\to} \top}\ \mathsf{aptm\_arrow\_top}$$

$$\frac{\Delta \vdash R_1 \trianglelefteq C \overset{\Rightarrow}{\to} S_1 \qquad \Delta \vdash R_2 \trianglelefteq C \overset{\Rightarrow}{\to} S_2}{\Delta \vdash (R_1 \,\&\, R_2) \trianglelefteq C \overset{\Rightarrow}{\to} (S_1 \,\&\, S_2)}\ \mathsf{aptm\_inter} \qquad \frac{}{\Delta \vdash \top \trianglelefteq C \overset{\Rightarrow}{\to} \top}\ \mathsf{aptm\_top}$$

2.10.3 Soundness and completeness

The following theorem shows that our algorithm is sound, by relating sort checking for an annotated term with the sorts assigned by the declarative system for the same annotated term.

Theorem 2.10.3 (Algorithmic sort-checking soundness)

1. *If* $\Delta \vdash C \stackrel{\Leftarrow}{\in} R$ *then* $\Delta \vdash C \in R$.
2. *If* $\Delta \vdash I \stackrel{\Rightarrow}{\in} R$ *then* $\Delta \vdash I \in R$.
3. *If* $\Delta \vdash R \trianglelefteq C \stackrel{\Rightarrow}{\to} S_2$ *then there exists* S_1 *such that* $\Delta \vdash C \in S_1$ *and* $R \leq S_1 \to S_2$.

Proof:
By induction on the given derivation.

The cases for rules sc_inter, sc_top, sc_lam, si_var and si_const simply apply the induction hypothesis to the premises, then rebuild the derivation using the corresponding sort assignment rule. The cases for the rules sc_atom, si_ann and si_app are similar, but slightly more complicated, so we show them below, along with the cases for the third part.

Case: $\dfrac{\Delta \vdash I \stackrel{\Rightarrow}{\in} S \qquad S \trianglelefteq R^{\mathsf{n}}}{\Delta \vdash I \stackrel{\Leftarrow}{\in} R^{\mathsf{n}}}$ sc_atom

$\Delta \vdash I \in S$	Ind. hyp.
$S \leq R^{\mathsf{n}}$	Correctness of $\trianglelefteq$ (2.9.6)
$\Delta \vdash I \in R^{\mathsf{n}}$	Rule sa_subs

Case: $\dfrac{\begin{array}{cccc}\Delta \vdash C \stackrel{\Leftarrow}{\notin} R_1 & \ldots & \Delta \vdash C \stackrel{\Leftarrow}{\notin} R_m \\ \Delta \vdash C \stackrel{\Leftarrow}{\in} S_1 & \ldots & \Delta \vdash C \stackrel{\Leftarrow}{\in} S_n\end{array}}{\Delta \vdash (C{\in}\{R_1, \ldots, R_m, S_1, \ldots, S_n\}) \stackrel{\Rightarrow}{\in} \&\{S_1, \ldots, S_n\}}$ si_ann

For each $i = 1, \ldots, n$:

$\Delta \vdash C \in S_i$	Ind. hyp.
$\Delta \vdash (C{\in}\{R_1, \ldots, R_m, S_1, \ldots, S_n\}) \in S_i$	Rule sa_ann
$\Delta \vdash (C{\in}\{R_1, \ldots, R_m, S_1, \ldots, S_n\}) \in \&\{S_1, \ldots, S_n\}$	Rule sa_inter

Case: $\dfrac{\Delta \vdash I \stackrel{\Rightarrow}{\in} R \qquad \Delta \vdash R \trianglelefteq C \stackrel{\Rightarrow}{\to} S}{\Delta \vdash I\, C \stackrel{\Rightarrow}{\in} S}$ si_app

$\Delta \vdash I \in R$	Ind. hyp.
$\Delta \vdash C \in S_1$ with $R \trianglelefteq S_1 \to S$	Ind. hyp.
$\Delta \vdash I \in S_1 \to S$	Rule sa_subs
$\Delta \vdash I\, C \in S$	Rule sa_app

Case: $\dfrac{\Delta \vdash C \stackrel{\Leftarrow}{\in} R_1}{\Delta \vdash R_1 \to R_2 \trianglelefteq C \stackrel{\Rightarrow}{\to} R_2}$ aptm_arrow

$\Delta \vdash C \in R_1$	Ind. hyp.
$R_1 \to R_2 \leq R_1 \to R_2$	Rule sub_reflex

As required, choosing $S_1 = R_1$.

Case: $\dfrac{\Delta \vdash C \overset{\Leftarrow}{\notin} R_1}{\Delta \vdash R_1 \to R_2 \trianglelefteq C \overset{\Rightarrow}{\to} \top}$ aptm_arrow_top

$\Delta \vdash C \in \top$	Rule sa_top
$R_1 \to R_2 \leq \top$	Rule sub_Tright
$R_1 \to R_2 \leq \top \to \top$	Rules sub_Tdist, sub_trans

Case: $\dfrac{\Delta \vdash R_1 \trianglelefteq C \overset{\Rightarrow}{\to} S_1 \qquad \Delta \vdash R_2 \trianglelefteq C \overset{\Rightarrow}{\to} S_2}{\Delta \vdash (R_1 \,\&\, R_2) \trianglelefteq C \overset{\Rightarrow}{\to} (S_1 \,\&\, S_2)}$ aptm_inter

$\Delta \vdash C \in S_1'$ with $R_1 \leq S_1' \to S_1$	Ind. hyp.
$\Delta \vdash C \in S_2'$ with $R_2 \leq S_2' \to S_2$	Ind. hyp.
$\Delta \vdash C \in S_1' \,\&\, S_2'$	Rule sa_inter
$R_1 \leq (S_1' \,\&\, S_2') \to S_1$	Rules sub_left1, sub_arrow, sub_reflex
$R_2 \leq (S_1' \,\&\, S_2') \to S_2$	Rules sub_left2, sub_arrow, sub_reflex
$R_1 \,\&\, R_2 \leq (S_1' \,\&\, S_2' \to S_1) \,\&\, (S_1' \,\&\, S_2' \to S_2)$	Rules sub_left1/2, sub_trans, sub_&right
$R_1 \,\&\, R_2 \leq (S_1' \,\&\, S_2') \to (S_1 \,\&\, S_2)$	sub_&dist

Case: $\dfrac{}{\Delta \vdash \top \trianglelefteq C \overset{\Rightarrow}{\to} \top}$ aptm_top

$\Delta \vdash C \in \top$	Rule sa_top
$\top \leq \top \to \top$	Rules sub_Tdist, sub_trans

□

The proof of completeness of the sort checking algorithm is somewhat more involved, since a declarative derivation may be structured in a way that can not be directly mirrored in an algorithmic derivation. We use the following *inversion lemmas* to assist in the required restructuring of derivations. Each essentially allows "inverting" one of the inference rules in the declarative system, i.e., it demonstrates that the premises of the rule must hold if the conclusion does. Similar properties are often immediate in simpler languages without subtyping and intersection types, but here they require explicit proofs due to the presence of rules that apply regardless of the form of the term, namely the rules sa_inter, sa_top and sa_subs.

Lemma 2.10.4 (Inversion for applications)
If $\Delta \vdash M\,N \in R$ then $\Delta \vdash M \in S \to R$ and $\Delta \vdash N \in S$ for some S.

Proof: By induction on the structure of the derivation of $\Delta \vdash M\,N \in R$.

Case: $\dfrac{\Delta \vdash M \in S \to R \qquad \Delta \vdash N \in S}{\Delta \vdash M\,N \in R}$ sa_app : Immediate.

Case: $\dfrac{\Delta \vdash M\,N \in R_1 \qquad \Delta \vdash M\,N \in R_2}{\Delta \vdash M\,N \in R_1 \& R_2}$ sa_inter

$\Delta \vdash M \in S_1 \to R_1$ and $\Delta \vdash N \in S_1$ for some S_1	Ind. hyp.

$\Delta \vdash M \in S_2 \to R_2$ and $\Delta \vdash N \in S_2$ for some S_2 — Ind. hyp.

$\Delta \vdash N \in S_1 \& S_2$ — Rule sa_inter

$\Delta \vdash M \in (S_1 \& S_2) \to R_1$ — Rule sa_subs via sub_arrow, sub_&left1 and sub_reflex

$\Delta \vdash M \in (S_1 \& S_2) \to R_2$ — Similarly

$\Delta \vdash M \in (S_1 \& S_2) \to (R_1 \& R_2)$ — Rule sub_&dist

Case: $\dfrac{}{\Delta \vdash M\ N \in \top}$ sa_top

$\Delta \vdash N \in \top$ — Rule sa_top

$\Delta \vdash M \in \top$ — Rule sa_top

$\Delta \vdash M \in \top \to \top$ — Rule sa_subs via sub_Tdist

Case: $\dfrac{\Delta \vdash M\ N \in R' \qquad R' \leq R}{\Delta \vdash M\ N \in R}$ sa_subs

$\Delta \vdash M \in S \to R'$ and $\Delta \vdash N \in S$ for some S — Ind. hyp.

$S \to R' \leq S \to R$ — Rule sub_arrow via sub_reflex

$\Delta \vdash M \in S \to R$ — Rule sa_subs

□

Lemma 2.10.5 (Inversion for $\to$ with $\leq$)
If $R_1 \to R_2 \leq S_1 \to S_2$ then either $\top \leq S_2$ or ($S_1 \leq R_1$ and $R_2 \leq S_2$).

Proof:
By inversion on the algorithmic subsorting derivation, using the Correctness of $\trianglelefteq$ (2.9.6).

□

Lemma 2.10.6 (Inversion for λ)
If $\Delta \vdash \lambda x.M \in R$ and $R \leq S_1 \to S_2$ then $\Delta, x{\in}S_1 \vdash M \in S_2$.

Proof: By induction on the structure of the derivation of $\Delta \vdash \lambda x.M \in R$.

Case: $\dfrac{\Delta, x{\in}R_1 \vdash M \in R_2}{\Delta \vdash \lambda x.M \in R_1 \to R_2}$ sa_lam

$R_1 \to R_2 \leq S_1 \to S_2$ — Assumption

$\top \leq S_2$ or ($S_1 \leq R_1$ and $R_2 \leq S_2$) — Inv. for $\to$ with $\leq$ (2.10.5)

If $\top \leq S_2$: the result follows by sa_top and sa_subs

Otherwise $S_1 \leq R_1$ and $R_2 \leq S_2$ and then:

$\Delta, x{\in}S_1 \vdash x \in S_1$ — Rule sa_var

$\Delta, x{\in}S_1 \vdash x \in R_1$ — Rule sa_subs

$\Delta, x{\in}S_1 \vdash M \in R_2$ — Substitution lemma

$\Delta, x{\in}S_1 \vdash M \in S_2$ — Rule sa_subs

Case: $\dfrac{\Delta \vdash \lambda x.M \in R_1 \qquad \Delta \vdash \lambda x.M \in R_2}{\Delta \vdash \lambda x.M \in R_1 \& R_2}\,\mathsf{sa\_inter}$

$R_1 \trianglelefteq S_1 \overset{\Rightarrow}{\to} S_{21}$ and $R_2 \trianglelefteq S_1 \overset{\Rightarrow}{\to} S_{22}$ with $S_{21}\&S_{22} \leq S_2$	Inv. on algorithmic subsorting
$\Delta, x{\in}S_1 \vdash M \in S_{21}$	Ind. hyp.
$\Delta, x{\in}S_1 \vdash M \in S_{22}$	Ind. hyp.
$\Delta, x{\in}S_1 \vdash M \in S_{21}\&S_{22}$	Rule sa_inter
$\Delta, x{\in}S_1 \vdash M \in S_2$	Rule sa_subs

Case: $\dfrac{}{\Delta \vdash \lambda x.M \in \top}\,\mathsf{sa\_top}$

$\top \leq S_1 \to S_2$	Assumption
$\top \leq S_2$	Inv. on algorithmic subsorting
$\Delta, x{\in}S_1 \vdash M \in \top$	Rule sa_top
$\Delta, x{\in}S_1 \vdash M \in S_2$	Rule sa_subs

Case: $\dfrac{\Delta \vdash \lambda x.M \in R' \qquad R' \leq R}{\Delta \vdash \lambda x.M \in R}\,\mathsf{sa\_subs}$

$R' \leq S_1 \to S_2$	Rule sub_trans
$\Delta, x{\in}S_1 \vdash M \in S_2$	Ind. hyp.

□

Lemma 2.10.7 (Inversion for annotations)
If $\Delta \vdash (M{\in}L) \in R$
then $\&\{S_1, \ldots, S_n\} \leq R$ *for some* $S_1, \ldots, S_n$ *with each* S_i *in* L *and also* $\Delta \vdash M \in S_i$.

Proof: By induction on the structure of the derivation of $\Delta \vdash (M{\in}L) \in R$.

Case: $\dfrac{R \text{ in } L \qquad \Delta \vdash M \in R}{\Delta \vdash (M{\in}L) \in R}\,\mathsf{sa\_annot}$

$\&\{R\} = R$	Def.
$R \leq R$	Rule sub_reflex

Case: $\dfrac{\Delta \vdash (M{\in}L) \in R_1 \qquad \Delta \vdash (M{\in}L) \in R_2}{\Delta \vdash (M{\in}L) \in R_1 \& R_2}\,\mathsf{sa\_inter}$

$\&\{S_1, \ldots, S_m\} \leq R_1$ with each S_i in L and also $\Delta \vdash M \in S_i$	Ind. hyp.
$\&\{S'_1, \ldots, S'_n\} \leq R_2$ with each S'_i in L and also $\Delta \vdash M \in S'_i$	Ind. hyp.
$\&\{S_1, \ldots, S_m, S'_1, \ldots, S'_n\} \leq R_1 \& R_2$	Rules sub_&left1/2, sub_&right

Case: $\dfrac{}{\Delta \vdash (M{\in}L) \in \top}\,\mathsf{sa\_top}$

$\&\{\} = \top$	Def.
$\top \leq \top$	Rule sub_top (or sub_reflex)

Case: $\dfrac{\Delta \vdash (M{\in}L) \in R' \qquad R' \leq R}{\Delta \vdash (M{\in}L) \in R}$ sa_subs

$\&\{S_1, \ldots, S_m\} \leq R'$ with each S_i in L and also $\Delta \vdash M \in S_i$ — Ind. hyp.

$\&\{S_1, \ldots, S_m\} \leq R$ — Rule sub_trans

□

Lemma 2.10.8 (Inversion for variables)
If $\Delta \vdash x \in R$ then $x{\in}S$ in Δ for some $S \leq R$.

Proof: By a straightforward induction the structure of the derivation of $\Delta \vdash x \in R$. □

Lemma 2.10.9 (Inversion for constants)
If $\Delta \vdash_\Sigma c \in R$ and $c{\in}S$ in Σ then $S \leq R$.

Proof: By a straightforward induction the structure of the derivation of $\Delta \vdash c \in R$. □

The following theorem demonstrates that our algorithm is complete for appropriately annotated terms.

Theorem 2.10.10 (Algorithmic sort-checking completeness)

1. *If $\Delta \vdash I \in S$ then there exists R such that $\Delta \vdash I \stackrel{\Rightarrow}{\in} R$ and $R \leq S$.*
2. *If $\Delta \vdash C \in R$ and $R \leq S$ then $\Delta \vdash C \stackrel{\Leftarrow}{\in} S$.*
3. *If $\Delta \vdash C \in S_1$ and $R \leq S_1 \to S_2$ then there exists R_2 such that $\Delta \vdash R \trianglelefteq C \stackrel{\Rightarrow}{=} R_2$ and $R_2 \leq S_2$.*

Proof:
By induction on the structure of terms, ordering so that part 3 may make use of part 2 and part 2 may make use of part 1 for the same term. The cases for each term construct in part 1 and part 2 make use of the above inversion lemmas.

We have the following cases for part 1.

Case: $\Delta \vdash x \in S$: using Inv. lemma for var. (2.10.8) and then si_var.

Case: $\Delta \vdash c \in S$: using Inv. lemma for constant (2.10.9) and then si_const.

Case: $\Delta \vdash I\,C \in S$

$\Delta \vdash I \in S_2 \to S$ and $\Delta \vdash C \in S_2$ for some S_2 — Inv. lemma for app. (2.10.4)

$\Delta \vdash I \stackrel{\Rightarrow}{\in} R_1$ for some $R_1 \leq S_2 \to S$ — Ind. hyp. (1)

$\Delta \vdash R_1 \trianglelefteq C \stackrel{\Rightarrow}{=} R$ for some $R \leq S$ — Ind. hyp. (3)

$\Delta \vdash I\,C \stackrel{\Rightarrow}{\in} R$ — Rule si_app

Case: $\Delta \vdash (C{\in}L) \in S$

$\&\{S_1, \ldots, S_n\} \leq S$ with each S_i in L and $\Delta \vdash C \in S_i$ — Inv. lemma for annot. (2.10.7)

$\Delta \vdash C \overset{\Leftarrow}{\in} S_i$ for each S_i — Ind. hyp. (2)

$L = \{S_1, \ldots S_n\} \cup L_1 \cup L_2$ with $\Delta \vdash C \overset{\Leftarrow}{\in} R_i$ for each R_i in L_1 and $\Delta \vdash C \overset{\Leftarrow}{\notin} R_i$ for each R_i in L_2 — $\overset{\Leftarrow}{\in}$ must succeed or fail (by termination)

$\Delta \vdash (C{\in}L) \overset{\Rightarrow}{\in} \&(\{S_1, \ldots S_n\} \cup L_1)$ — Rule si_ann

$\&(\{S_1, \ldots S_n\} \cup L_1) \leq \&\{S_1, \ldots S_n\}$ — Rule sub_&left1

$\&(\{S_1, \ldots S_n\} \cup L_1) \leq S$ — Rule sub_trans

Part 2 requires a sub-induction on the structure of the sort R to account for intersections. We treat the cases for intersection and top first, then consider the remaining cases based on the term C.

Case: $\Delta \vdash C \in S_1 \& S_2$

$\Delta \vdash C \in S_1$ — Rule sa_subs via sub_&left1

$\Delta \vdash C \overset{\Leftarrow}{\in} S_1$ — Ind. hyp. (2)

$\Delta \vdash C \overset{\Leftarrow}{\in} S_2$ — Similarly

$\Delta \vdash C \overset{\Leftarrow}{\in} S_1 \& S_2$ — Rule sc_inter

Case: $\Delta \vdash C \in \top$

$\Delta \vdash C \overset{\Leftarrow}{\in} \top$ — Rule sc_top

Case: $\Delta \vdash \lambda x.C \in S_1 \to S_2$

$S_1 \to S_2 \leq S_1 \to S_2$ — Rule sub_reflex

$\Delta, x{\in}S_1 \vdash C \in S_2$ — Inv. for λ (2.10.6)

$\Delta, x{\in}S_1 \vdash C \overset{\Leftarrow}{\in} S_2$ — Ind. hyp. (2)

$\Delta \vdash \lambda x.C \overset{\Leftarrow}{\in} S_1 \to S_2$ — Rule sc_lam

Case: $\Delta \vdash I \in S^{\mathsf{n}}$

$\Delta \vdash I \overset{\Rightarrow}{\in} R$ for some $R \leq S^{\mathsf{n}}$ — Ind. hyp. (1)

$\Delta \vdash I \overset{\Leftarrow}{\in} S^{\mathsf{n}}$ — Rule sc_atom

Part 3 also requires a sub-induction on the structure of the sort R to account for intersections. We have the following cases.

Case: $R_1 \& R_2 \leq S_1 \to S_2$ and $\Delta \vdash C \in S_1$

$R_1 \trianglelefteq S_1 \overset{\Rightarrow}{\to} S_{21}$ and $R_2 \trianglelefteq S_1 \overset{\Rightarrow}{\to} S_{22}$ with $S_{21} \& S_{22} \leq S_2$ — Inv. on algorithmic subsorting

$R_1 \leq S_1 \to S_{21}$ — Correctness of $\trianglelefteq$ (2.9.6)

$\Delta \vdash R_1 \trianglelefteq C \overset{\Rightarrow}{\to} R_{21}$ for some $R_{21} \leq S_{21}$ — Ind. hyp. (3)

$\Delta \vdash R_2 \trianglelefteq C \overset{\Rightarrow}{\to} R_{22}$ for some $R_{22} \leq S_{22}$ — Similarly

$\Delta \vdash R_1 \& R_2 \trianglelefteq C \overset{\Rightarrow}{\to} R_{21} \& R_{22}$ — Rule aptm_inter

$R_{21} \& R_{22} \leq S_{21} \& S_{22}$ — Rules sub_&left1/2, sub_right

$S_{21} \& S_{22} \leq S_2$ — Rule sub_trans

Case: $\top \leq S_1 \to S_2$ and $\Delta \vdash C \in S_1$

$\top \leq S_2$ — Inv. on algorithmic subsorting

$\Delta \vdash \top \trianglelefteq C \Rightarrow \top$ — Rule aptm_top

Case: $R_1 \to R_2 \leq S_1 \to S_2$ and $\Delta \vdash C \in S_1$

$\top \leq S_2$ or ($S_1 \leq R_1$ and $R_2 \leq S_2$) — Inv. for $\to$ with $\leq$ (2.10.5)

If $\top \leq S_2$: the result follows by aptm_arrow or aptm_arrow_top, then sub_top.

Otherwise $S_1 \leq R_1$ and $R_2 \leq S_2$ and then:

$\Delta \vdash C \in R_1$ — Rule sa_subs

$\Delta \vdash C \Leftarrow R_1$ — Ind. hyp. (1)

$\Delta \vdash R_1 \to R_2 \trianglelefteq C \Rightarrow R_2$ — Rule aptm_arrow

□

Together the above soundness and completeness results imply that the sort checking algorithm and the declarative sort assignment judgment (extended with the rule for annotations) exactly agree on the sorts that may be assigned to a term, provided that the term meets the grammar for an inferable or checkable term.

2.11 Annotatability

We now demonstrate that terms can always be annotated to obtain a desired sort, provided that the unannotated term has that sort. Roughly, we want to prove a theorem like the following.

> If $\Delta \vdash M \in R$ then we can construct a checkable term C such that $\|C\| = M$ and $\Delta \vdash C \in R$ (and similarly for inferable terms).

However, a difficulty arises when we attempt to prove such a theorem using a standard structural induction on sorting derivations. The main difficulty arises when we have a sorting derivation of the following form.

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash M \in R_1 & \Delta \vdash M \in R_2\end{array}}{\Delta \vdash M \in R_1 \,\&\, R_2}\ \mathsf{sa\_inter}$$

Applying the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ yields two terms C_1 and C_2 such that $\|C_1\| = M$, $\|C_2\| = M$, $\Delta \vdash C_1 \in R_1$ and $\Delta \vdash C_2 \in R_2$. But we cannot apply the rule sc_inter since C_1 and C_2 may be different terms.

One possible solution to this difficulty is to generalize the induction hypothesis so that it directly produces a single term from multiple sorting derivations, i.e. roughly like the following, which would require induction on the sum of the sizes of the assumed derivations.

> If $\Delta_1 \vdash M \in R_1$ and $\Delta_2 \vdash M \in R_2$ and ... and $\Delta_n \vdash M \in R_n$ then we can construct C such that $\|C\| = M$ and
> $\Delta_1 \vdash C \in R_1$ and $\Delta_2 \vdash C \in R_2$ and ... and $\Delta_n \vdash C \in R_n$

It appears that this approach would succeed, but the use of lists would be somewhat notationally awkward.

Here we will present another solution to the above problem by showing how to construct a term C which combines the annotations from C_1 and C_2, such that $\Delta \vdash C \in R_1$ and $\Delta \vdash C \in R_2$.

In doing so, we wish to avoid some complications that arise in the case where C_1 and C_2 have annotations on different subterms. We thus restrict our attention to checkable and inferable terms with annotations on exactly those subterms where they cannot be avoided. We call such terms "minimal checkable terms" and "minimal inferable terms" to capture the intuition that they only have annotations where required by the definitions of inferable and checkable terms. An alternative would be to require annotations on every subterm, but it seems preferable to work with a minimum of annotations.

Definition 2.11.1

$$\begin{array}{rrl} \textit{Minimal Inferable Terms} & I^{\mathsf{m}} & ::= c \mid x \mid I^{\mathsf{m}}\, C^{\mathsf{m}} \mid ((\lambda x{:}A.C^{\mathsf{m}}) \in L) \\ \textit{Minimal Checkable Terms} & C^{\mathsf{m}} & ::= c \mid x \mid I^{\mathsf{m}}\, C^{\mathsf{m}} \mid \lambda x{:}A.C^{\mathsf{m}} \end{array}$$

The following function $C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}$ combines the annotations from two minimal checkable terms C_1^{m} and C_2^{m} which satisfy $\|C_1^{\mathsf{m}}\| = \|C_2^{\mathsf{m}}\|$. It uses an similar function $I_1^{\mathsf{m}} \overset{\mathsf{i}}{\bowtie} I_2^{\mathsf{m}}$ to combine two minimal inferable terms satisfying $\|I_1^{\mathsf{m}}\| = \|I_2^{\mathsf{m}}\|$. These functions are defined inductively following the definitions of minimal inferable terms and minimal checkable terms. The constraints $\|C_1^{\mathsf{m}}\| = \|C_2^{\mathsf{m}}\|$ and $\|I_1^{\mathsf{m}}\| = \|I_2^{\mathsf{m}}\|$ guarantee that the two terms are identical other than the choice of sort annotations.

$$\begin{array}{rcl} c \overset{\mathsf{i}}{\bowtie} c & = & c \\ x \overset{\mathsf{i}}{\bowtie} x & = & x \\ (I_1^{\mathsf{m}}\, C_1^{\mathsf{m}}) \overset{\mathsf{i}}{\bowtie} (I_2^{\mathsf{m}}\, C_2^{\mathsf{m}}) & = & (I_1^{\mathsf{m}} \overset{\mathsf{i}}{\bowtie} I_2^{\mathsf{m}})\, (C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}) \\ ((\lambda x{:}A.C_1^{\mathsf{m}}) \in L_1) \overset{\mathsf{i}}{\bowtie} ((\lambda x{:}A.C_2^{\mathsf{m}}) \in L_2) & = & ((\lambda x{:}A.C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}) \in L_1, L_2) \end{array}$$

$$\begin{array}{rcl} c \overset{\mathsf{c}}{\bowtie} c & = & c \\ x \overset{\mathsf{c}}{\bowtie} x & = & x \\ (I_1^{\mathsf{m}}\, C_1^{\mathsf{m}}) \overset{\mathsf{c}}{\bowtie} (I_2^{\mathsf{m}}\, C_2^{\mathsf{m}}) & = & (I_1^{\mathsf{m}} \overset{\mathsf{i}}{\bowtie} I_2^{\mathsf{m}})\, (C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}) \\ (\lambda x{:}A.C_1^{\mathsf{m}}) \overset{\mathsf{c}}{\bowtie} (\lambda x{:}A.C_2^{\mathsf{m}}) & = & (\lambda x{:}A.C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}) \end{array}$$

These definitions mostly just follow the shared structure of the terms. The only interesting case is when we have inferable terms which are abstractions, in which case both terms must have annotations, and we concatenate the two annotations (the notation L_1, L_2 means the concatenation of the two lists).

The following lemma demonstrates that $\overset{\mathsf{c}}{\bowtie}$ and $\overset{\mathsf{i}}{\bowtie}$ have the intended properties.

Lemma 2.11.2 (Annotation Combination)

1. *If* $\|I_1^{\mathsf{m}}\| = \|I_2^{\mathsf{m}}\|$ *and either* $\Delta \vdash I_1^{\mathsf{m}} \in R$ *or* $\Delta \vdash I_2^{\mathsf{m}} \in R$ *then*
   $\Delta \vdash (I_1^{\mathsf{m}} \overset{\mathsf{i}}{\bowtie} I_2^{\mathsf{m}}) \in R$.

2. *If* $\|C_1^{\mathsf{m}}\| = \|C_2^{\mathsf{m}}\|$ *and either* $\Delta \vdash C_1^{\mathsf{m}} \in R$ *or* $\Delta \vdash C_2^{\mathsf{m}} \in R$ *then* $\Delta \vdash (C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}) \in R$.

Proof: (sketch)
By structural induction on the sorting derivations. All cases are straightforward. □

Theorem 2.11.3 (Annotatability)
If $\Delta \vdash M \in R$ *then we can construct a minimal inferable term* I^{m} *and a minimal checkable term* C^{m} *such that* $\|I^{\mathsf{m}}\| = \|C^{\mathsf{m}}\| = M$ *and* $\Delta \vdash I^{\mathsf{m}} \in R$ *and* $\Delta \vdash C^{\mathsf{m}} \in R$.

Proof: By induction on the sorting derivation. We show two cases. The remaining cases simply rebuild the term, using the induction hypothesis on subderivations.

Case:
$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash M \in R_1 & \Delta \vdash M \in R_2\end{array}}{\Delta \vdash M \in R_1 \,\&\, R_2}\ \mathsf{sa\_inter}$$

If $M = c$ or $M = x$ then M already has the required forms, so we set $I^{\mathsf{m}} = M$ and $C^{\mathsf{m}} = M$.
Otherwise $M = N_1\, N_2$ or $M = \lambda x{:}A.M_1$. We show the latter case only: the former is similar.

$\Delta \vdash C_1^{\mathsf{m}} \in R_1$	and	
$\|C_1^{\mathsf{m}}\| = M$	for some C_1^{m}	By ind. hyp. on $\mathcal{D}_1$
$\Delta \vdash C_2^{\mathsf{m}} \in R_2$	and	
$\|C_2^{\mathsf{m}}\| = M$	for some C_2^{m}	By ind. hyp. on $\mathcal{D}_2$
$\Delta \vdash C^{\mathsf{m}} \in R_1$	and	
$\Delta \vdash C^{\mathsf{m}} \in R_2$	and	
$\|C^{\mathsf{m}}\| = M$	for $C^{\mathsf{m}} = C_1^{\mathsf{m}} \overset{\mathsf{c}}{\bowtie} C_2^{\mathsf{m}}$	By above lemma
$\Delta \vdash C^{\mathsf{m}} \in R_1 \,\&\, R_2$		By rule sa_inter
$\Delta \vdash (C^{\mathsf{m}} \in R_1 \,\&\, R_2) \in R_1 \,\&\, R_2$		By rule sa_annot
$\|(C^{\mathsf{m}} \in R_1 \,\&\, R_2)\| = \|C^{\mathsf{m}}\| = M$		By defn of $\|\cdot\|$

Then C^{m} is as required, and we set $I^{\mathsf{m}} = (C^{\mathsf{m}} \in R_1 \,\&\, R_2)$.

Case:
$$\frac{\begin{array}{c}\mathcal{D}_2 \\ \Delta, x{\in}R_1 \vdash M_2 \in R_2\end{array}}{\Delta \vdash \lambda x{:}A.M_2 \in R_1 \to R_2}\ \mathsf{sa\_lam}$$

$\Delta, x{\in}R_1 \vdash C_2^{\mathsf{m}} \in R_2$	and	
$\|C_2^{\mathsf{m}}\| = M_2$	for some C_2^{m}	By ind. hyp. on $\mathcal{D}_2$
$\Delta \vdash \lambda x{:}A.C_2^{\mathsf{m}} \in R_1 \to R_2$		By rule sa_lam
$\Delta \vdash ((\lambda x{:}A.C_2^{\mathsf{m}}) \in R_1 \to R_2) \ \in\ R_1 \to R_2$		By rule sa_annot

Then

$$\|(\lambda x{:}A.C_2^{\mathsf{m}}) \in R_1 \to R_2)\| = \|\lambda x{:}A.C_2^{\mathsf{m}}\| = \lambda x{:}A.\|C_2^{\mathsf{m}}\| = \lambda x{:}A.M_2$$

and we choose $C^{\mathsf{m}} = \lambda x{:}A.C_2^{\mathsf{m}}$ and $I^{\mathsf{m}} = ((\lambda x{:}A.C_2^{\mathsf{m}}) \in R_1 \to R_2)$.

□

Chapter 3

Sort checking with a value restriction

Our first major challenge in extending sorts to Standard ML is that standard formulations of intersection type systems are unsound in the presence of call-by-value computational effects. We recall the following example from Section 1.4.2 which demonstrates the unsoundness that results when we naïvely extend sorts to the reference types of ML.

```
(*[ cell <: (nat ref) & (pos ref) ]*)
val cell = ref one
val () = (cell := zero)

(*[ result <: pos ]*)
val result = !cell
```

Here we create a reference `cell` containing `one`. Now, we can assign two sorts to `ref one`, as follows: `one` has sort `pos` so `ref one` has sort `pos ref`, and `one` has sort `nat` so `ref one` has sort `nat ref`. The standard rule for the introduction of intersection (sa_inter in Chapter 2) then allows us to assign the intersection of these sorts: `cell <: (pos ref) & (nat ref)`. But this is unsound, because the second part of this sort allows us to update `cell` with `zero`, while the first part requires that reading from the cell only returns values satisfying `pos`.

The introduction rule for top (sa_top in Chapter 2) is also unsound in the presence of effects, since it allows us to assign the sort $\top$ to any expression whatsoever, including one that performs an inappropriate update on a reference cell, such as the following.

```
(*[ cell <: pos ref ]*)
val cell = ref one

(*[ un <: top[unit] ]*)
val un = (cell := zero)

(*[ result <: pos ]*)
val result = !cell
```

This chapter presents a solution to these problems, using a new variant of intersection types that are sound in the presence of effects. This is achieved via a *value restriction* on the introduction

of intersection polymorphism. Our restriction is similar to the value restriction on parametric polymorphism proposed by Wright [Wri95] and included in the revised definition of Standard ML [MTHM97] to prevent unsound uses of parametric polymorphism.

It is not immediately clear that we can add such a restriction on intersections and still have a sensible system: some basic properties might fail to hold, such as preservation of sorts during reduction. The situation is more complicated than the case of parametric polymorphism in ML because we have a rich set of inclusions between sorts. Additionally, the following example shows that introducing a value restriction on intersections is not enough by itself to guarantee soundness in the presence of effects.

```
(*[ f <: (unit -> pos ref) & (unit -> nat ref) ]*)
fun f () = ref one

(*[ cell <: (pos ref) & (nat ref) ]*)
val cell = f ()
```

This example is unsound for similar reasons to the first example. Here the sort for `cell` is obtained by using subsumption on the sort for `f` with an instance of the standard distributivity subtyping rule for intersection types. This is the rule sub_&dist from Chapter 2, which is as follows.

$$(R \to S_1) \,\&\, (R \to S_2) \;\leq\; R \to (S_1 \,\&\, S_2)$$

We can analyze this situation by considering an effectful function $R \to S$ as equivalent to a pure function ($\Rightarrow$) that returns an "effectful computation" $R \Rightarrow \bigcirc S$, as in the computational meta-language proposed by Moggi [Mog89, Mog91]. Then, the following subsorting is sound, because it is a substitution instance of the standard distributivity rule for pure functions.

$$(R \Rightarrow \bigcirc S_1) \,\&\, (R \Rightarrow \bigcirc S_2) \;\leq\; R \Rightarrow (\bigcirc S_1 \,\&\, \bigcirc S_2)$$

However, the unsound distributivity rule for effectful functions corresponds to the following.

$$(R \Rightarrow \bigcirc S_1) \;\&\; (R \Rightarrow \bigcirc S_2) \;\leq\; R \Rightarrow \bigcirc(S_1 \,\&\, S_2)$$

This is unsound because in general $(\bigcirc S_1)\&(\bigcirc S_2)$ is not a subsort of $\bigcirc(S_1 \& S_2)$. For example, if we make effectful computations explicit using $\bigcirc$ then we should be able to assign the following sort to `ref one`.

$$\texttt{ref one} \;\in\; \bigcirc(\texttt{pos ref}) \;\&\; \bigcirc(\texttt{nat ref})$$

However, we should not be able to assign the following sort.

$$\texttt{ref one} \;\in\; \bigcirc((\texttt{pos ref}) \;\&\; (\texttt{nat ref}))$$

Roughly this is because the computation creates a reference cell when it executes, and we can designate that the reference cell contains values with sort `pos`, or we can designate that the cell contains values with sort `nat`, but it inconsistent to designate both. This explanation applies both to the unsoundness of the distributivity rule and to the unsoundness of intersections without a value restriction.

For similar reasons, the standard rule for distributivity of $\top$ is also unsound. This is the rule sub_Tdist in Chapter 2). The other standard subtyping rules for intersection and function types (see Section 2.4) are sound with call-by-value effects. Roughly, this is because none of them involve both functions and intersections, and so none of them require inclusions like $(\circ S_1) \,\&\, (\circ S_2) \leq \circ(S_1 \,\&\, S_2)$ to be sound when effectful functions are added.

We thus discard the two distributivity rules, and retain the other rules. This leaves a subsorting system with a pleasing orthogonality between the rules: each rule only involves a single sort construct.

The main purpose of this chapter is to consider sorts with a value restriction and no distributivity in a simple and general context, and to present a bidirectional checking algorithm for them. We thus focus on a λ-calculus with these features, and with arbitrary finite lattices of refinements for each base type. We call this language $\lambda_v^{\rightarrow\&}$.

Unlike Chapter 2 we directly assume that the refinements of base types form finite lattices: this allows the presentation of the algorithm to more accurately reflect the intended implementation. The more abstract declarations of base refinements considered in Chapter 2 were proven equivalent to finite lattices in Section 2.6, and this result does not depend on the distributivity rules. Thus, this equivalence also relates the base lattices in this chapter to the more abstract declarations. This roughly corresponds with what we do in our implementation: externally there may be many refinements of a base type that are equivalent, but internally refinements of base types are represented using a canonical representative for each equivalence class.

Outline of this chapter

The development in this chapter is broadly similar to that in Chapter 2. We generally give more technical details of proofs in this chapter, since the formulation in this chapter is more central to this work than those in Chapter 2. Also, proofs of results for standard intersection types have often appeared elsewhere.

We begin by presenting the syntax for $\lambda_v^{\rightarrow\&}$, and standard judgments for typing, reduction, and refinement (Sections 3.1, 3.2, 3.3). One critical definition here is that of values. We include abstractions and variables as values, as might be expected. We also include applications of constants to values: these correspond to applications of constructors in ML.

We then state our assumptions for base lattices (Section 3.4), and present a judgment which defines subsorting for $\lambda_v^{\rightarrow\&}$ (Section 3.5). We then prove finiteness of refinements up to equivalence (Section 3.6), a result for which we omitted a proof in Chapter 2, in part because it follows from the result in this chapter, and in part because it has been considered by Freeman [Fre94] for the standard form of intersection types.

We then present our algorithm for determining subsorting, and prove it correct (Section 3.7). The lack of distributivity results in a simpler algorithm than in Chapter 2, and the proof is also simpler, and more structural. Next we present the sort assignment rules for $\lambda_v^{\rightarrow\&}$, and prove some basic results such as subject reduction with respect to sorts (Section 3.8).

We continue by considering some issues related to the sorts for applications of constants (Section 3.9), and presenting proofs of the existence of principal sorts, and the theoretical decidability of sort inference (Section 3.10). In the previous chapter corresponding proofs were omitted, or few details were provided, because similar results have been considered previously by Freeman [Fre94] for the standard form of intersection types.

Next we present our algorithm for bidirectional sort checking for $\lambda_v^{\rightarrow\&}$ (Section 3.11). This is the central algorithm in this dissertation, and forms the core of our implementation of a practical sort checker for Standard ML. We present a detailed proof the correctness of this algorithm with respect to the sort assignment rules. The lack of distributivity results in the algorithm and its proof being simpler than in the previous chapter, although it also results in the need for the algorithm to backtrack.

We conclude with a proof that shows that terms can be annotated appropriately to satisfy the bidirectional checking algorithm (Section 3.12). This proof is a small variation of that in the previous chapter.

In this chapter we do not include any effects. In Chapter 4 we formally demonstrate the soundness of the extension of $\lambda_v^{\rightarrow\&}$ to a call-by-value language with a standard feature involving effects, namely mutable references. It is our intention that $\lambda_v^{\rightarrow\&}$ could also serve as the basis for other languages with intersection types but without the distributivity subtyping rule. For example, sorts for LF have been considered by Pfenning [Pfe01a] that avoid distributivity so that the subsorting relation is more orthogonal and robustly extends to a variety of function spaces, including function spaces involving modalities such as those in linear logic.

3.1 Syntax

We now present the syntax for $\lambda_v^{\rightarrow\&}$. We separate types and sorts into separate syntactic classes, following Freeman [Fre94]. We use a signature Σ to capture typing and sorting assumptions for constants. However, we make a global assumption that there are some base types and base sorts that refine them. This is necessary to accommodate the approach to base lattices used in this chapter: we will be globally assuming that each base type has a lattice of refinements (although we delay formally introducing this assumption until Section 3.4).

$$
\begin{array}{rrl}
\text{Types} & A, B & ::= a \mid A_1 \rightarrow A_2 \\
\text{Type Contexts} & \Gamma & ::= \cdot \mid \Gamma, x{:}A \\
\text{Terms} & M, N & ::= c \mid x \mid \lambda x{:}A.M \mid M_1\, M_2 \\
\text{Sorts} & R, S & ::= r^a \mid R_1 \rightarrow R_2 \mid R_1 \,\&\, R_2 \mid \top^A \\
\text{Sort Contexts} & \Delta & ::= \cdot \mid \Delta, x{\in}R \\
\text{Signatures} & \Sigma & ::= \cdot \mid \Sigma, c{:}A \mid \Sigma, c{\in}R
\end{array}
$$

As in the previous chapter, we use a, b for base types and c for constants. We write $\{N/x\}M$ for the result of substituting N for the variable x in M, renaming bound variables as necessary to avoid the capture of free variables in N. We use the notation $\mathcal{D} :: J$ to indicate that $\mathcal{D}$ is a derivation of judgment J.

In this chapter we write base sorts as r^a, s^b so that the type being refined is inherent in the syntax. This replaces the declarations of the form $r \sqsubset a$ used in signatures in the previous chapter. This notation can result in excessive clutter, so we often elide the base types a, b when they are clear from context, or when they are of little interest.

3.2 Validity judgments

We now present the validity judgments for $\lambda_v^{\to\&}$. These are simpler than in the previous chapter, since base types and base sorts are not declared in signatures.

Valid refinements

First, we have the validity judgment for refinements. Unlike the previous chapter, this does not depend on the signature.

$R \sqsubset A$ Sort R is a well-formed sort refining type A.

$$\frac{}{r^a \sqsubset a}\ \mathsf{rf\_base} \qquad \frac{R \sqsubset A \qquad S \sqsubset B}{R \to S \sqsubset A \to B}\ \mathsf{rf\_arrow}$$

$$\frac{R \sqsubset A \qquad S \sqsubset A}{R \,\&\, S \sqsubset A}\ \mathsf{rf\_conj} \qquad \frac{}{\top^A \sqsubset A}\ \mathsf{rf\_top}$$

A direct consequence of this definition is that every sort refines at most one type. We extend the refinement notation pointwise to $\Delta \sqsubset \Gamma$ for sort context Δ and type context Γ which bind the same variables.

As in the previous chapter, we say that a sort R is *well-formed* if it refines some type A. In what follows, we will only be interested in well-formed sorts, and so when we use the term "sort" we implicitly mean "well-formed sort". We say that two sorts are *compatible* if they refine the same type. We say that a sort S is a *conjunct* of sort R if $S = R$, or (inductively) if $R = R_1 \,\&\, R_2$ and S is a conjunct of either R_1 or R_2.

Valid signatures

Next, we have the validity judgment for signatures. This only needs to check that constants are assigned appropriate sorts. In addition to checking that the sort R for a constant refines the right type, we also check that it is *distributive* which is defined as follows.

Definition 3.2.1
We say that the sort $R \sqsubset A_1 \to \ldots \to A_n$ is distributive if the following hold.

1. *Whenever $R \leq R_1 \to \ldots \to R_n$ and $R \leq S_1 \to \ldots \to S_n$ we have $R \leq (R_1 \,\&\, S_1) \to \ldots \to (R_n \,\&\, S_n)$.*
2. *$R \leq \top^{A_1} \to \ldots \to \top^{A_n}$.*

The distributivity check is required because applications of constructors are included as values, hence constructors should to be treated as pure functions, and so a form of distributivity should hold for them. Without such distributivity certain technical complications would arise in some of the proofs which follow. Essentially the check forces a form of distributivity for constructors to hold by requiring that their sorts be rich enough to compensate for the lack of the distributivity rule. The exact reasons for choosing this approach are somewhat technical,

and we delay full details until Section 3.9. For now we note that every sort can be "completed" into a distributive one, so the distributivity requirement is not a major restriction. Further, the constructor sorts which arise in our implementation naturally satisfy this condition. In other situations, this condition can be mechanically checked using the algorithmic form of subsorting which follows later in this chapter (it suffices to check each pair of conjuncts of R for the first part, rather than all pairs of supersorts).

Since the definition of distributivity depends on subsorting, this check results in the validity judgment depending on the subsorting judgment, but we have chosen to present the validity judgment first anyway, for consistency with the previous chapter.

$\vdash \Sigma\ \mathit{Sig}$ $\quad \Sigma$ is a valid signature

$$\frac{}{\vdash \cdot\ \mathit{Sig}}\ \mathsf{sigemp} \qquad \frac{\vdash \Sigma\ \mathit{Sig}}{\vdash \Sigma, c{:}A\ \mathit{Sig}}\ \mathsf{sigobj}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \qquad \vdash_\Sigma c : A \qquad \vdash_\Sigma R \sqsubset A \qquad \vdash_\Sigma R\ \mathit{distributive}}{\vdash \Sigma, c{\in}R\ \mathit{Sig}}\ \mathsf{sigobjsrt}$$

As an example of a valid signature, the following represents the types for the bit strings examples in the introduction: we have a type a_{bits} with positive and natural numbers as refinements r_{nat}, r_{pos} as well as a refinement r_{bits} that includes all bit strings. In this chapter we globally assume that we have some base types that each have a finite lattice of refinements, hence the signature does not declare a_{bits}, r_{nat}, r_{pos} and r_{bits}, nor does it declare inclusions such as $r_{\mathsf{pos}} \leq r_{\mathsf{nat}} \leq r_{\mathsf{bits}}$.

$$\begin{array}{lcl}
c_{\mathsf{bnil}} & : & a_{\mathsf{bits}}, \\
c_{\mathsf{b0}} & : & a_{\mathsf{bits}} \to a_{\mathsf{bits}}, \\
c_{\mathsf{b1}} & : & a_{\mathsf{bits}} \to a_{\mathsf{bits}}, \\
c_{\mathsf{bnil}} & \in & r_{\mathsf{nat}}, \\
c_{\mathsf{b0}} & \in & (r_{\mathsf{bits}} \to r_{\mathsf{bits}}) \,\&\, (r_{\mathsf{pos}} \to r_{\mathsf{pos}}) \\
c_{\mathsf{b1}} & \in & (r_{\mathsf{bits}} \to r_{\mathsf{bits}}) \,\&\, (r_{\mathsf{nat}} \to r_{\mathsf{pos}})
\end{array}$$

Valid terms

The typing judgment for $\lambda_v^{\to\&}$ has the following (standard) definition:

$\Gamma \vdash_\Sigma M : A$ $\quad$ Term M has type A in context Γ with signature Σ.

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash_\Sigma c : A}\ \mathsf{tp\_const} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : A}\ \mathsf{tp\_var}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.M : A \to B}\ \mathsf{tp\_lam} \qquad \frac{\Gamma \vdash_\Sigma M : A \to B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : B}\ \mathsf{tp\_app}$$

3.3 Values and reduction

Reduction for $\lambda_v^{\to\&}$ is defined using the standard β-value rule (following Plotkin [Plo75]). We choose here to focus on reduction rather than an operational semantics because the development in this chapter is intended to be suitable for a wide range of languages, including those that naturally include reduction within the body of λ-abstractions (such as LF).

First, we need to distinguish those terms which are values.

$$\begin{array}{lrl} \text{Atoms} & V^{\mathsf{a}} & ::= c \mid V^{\mathsf{a}}\, V \\ \text{Values} & V & ::= V^{\mathsf{a}} \mid x \mid \lambda x{:}A.M \end{array}$$

Atoms are repeated applications of constants to values. They correspond to applications of constructors in ML. While ML only allows a single application, we allow many applications so that a constructed value can be formed from many arguments. In ML products would be used for the same purpose, but $\lambda_v^{\to\&}$ does not have products. For example, the following signature includes type and sort constants that allow the construction of lists of elements which have type a_{elem} and sort $r_{\mathsf{elem}} \sqsubset a_{\mathsf{elem}}$.

$$\begin{array}{lcl} c_{\mathsf{nil}} & : & a_{\mathsf{list}}, \\ c_{\mathsf{cons}} & : & a_{\mathsf{elem}} \to a_{\mathsf{list}} \to a_{\mathsf{list}}, \\ c_{\mathsf{nil}} & \in & r_{\mathsf{list}}, \\ c_{\mathsf{cons}} & \in & r_{\mathsf{elem}} \to r_{\mathsf{list}} \to r_{\mathsf{list}} \end{array}$$

There are a number of other sensible definitions of values. The essential feature in the development that follows is that there is a distinguished class of terms to which we restrict the intersection introduction rule. We will point out interesting variations during this development, in particular in Section 3.8. In the context of effectful functions, values are those terms for which the absence of effects is syntactically immediate.

We have the following standard β-value reduction rule. It allows an argument to be substituted into the body of a function, provided that the argument is a value.

$$\frac{}{(\lambda x{:}A.M)\, V \mapsto \{V/x\}M}\ \beta_v$$

We also have the following compositional rules.

$$\frac{M \mapsto N}{\lambda x{:}A.M \mapsto \lambda x{:}A.N}\ \mathsf{reduce\_lam} \qquad \frac{M \mapsto M'}{M\, N \mapsto M'\, N}\ \mathsf{reduce\_app1} \qquad \frac{N \mapsto N'}{M\, N \mapsto M\, N'}\ \mathsf{reduce\_app2}$$

So far $\lambda_v^{\to\&}$ is a standard call-by-value λ-calculus, and we omit the proofs of standard results such as subject reduction with respect to types. Our main interest is in the refinements of the standard types.

3.4 Base sort lattices

To account smoothly for intersections of base sorts, we assume that there is a finite lattice of base sorts refining each base type. This is quite different from the finite lattices of equivalence classes in Chapter 2, which included sorts containing syntactic intersections of the form $r_1 \,\&\, \ldots \,\&\, r_n$.

In this chapter we have not yet considered subsorting and equivalence, and each element of one of these lattices is simply a base sort r^a, and we assume that we have the appropriate lattice operations, including a greatest lower bound operator that defines the intersection of two compatible base sorts as being some base sort refining the same type.

Formally, for each base type a we assume that the following are defined.

- A finite set of base sorts $r_1^a, \ldots, r_n^a$ that are the refinements of a.
- A binary operator $\overset{a}{\&}$ which defines the greatest lower bound operation of the lattice, and maps each pair of base sorts r_1^a, r_2^a to a base sort r_3^a.
- $\overset{a}{\top}$ which is a distinguished base sort r^a that is the top element of the lattice.
- A partial order $\overset{a}{\leq}$ defined on pairs of base sorts that refine the same base type a.

The least upper bounds in the lattice are not explicitly considered here: they can be defined as the (finite) intersection of all upper bounds. We further assume that $\overset{a}{\&}$, $\overset{a}{\top}$ and $\overset{a}{\leq}$ satisfy the following conditions (which are based on Freeman [Fre94]). Each of these is implicitly universally quantified over all base sorts refining the appropriate type.

Assumption 1 (base reflexivity) $r \overset{a}{\leq} r$.

Assumption 2 (base transitivity) *If* $r_1 \overset{a}{\leq} r_2$ *and* $r_2 \overset{a}{\leq} r_3$ *then* $r_1 \overset{a}{\leq} r_3$.

Assumption 3 ($\overset{a}{\&}$ is lower bound) $r \overset{a}{\&} s \overset{a}{\leq} r$ *and* $r \overset{a}{\&} s \overset{a}{\leq} s$.

Assumption 4 ($\overset{a}{\&}$ is maximal) *If* $s \overset{a}{\leq} r_1$ *and* $s \overset{a}{\leq} r_2$ *then* $s \overset{a}{\leq} r_1 \overset{a}{\&} r_2$.

Assumption 5 ($\overset{a}{\top}$ is maximal) $r \overset{a}{\leq} \overset{a}{\top}$.

One method of defining such base sort lattices is via the declarations used in Chapter 2. Then, by the equivalence to finite lattices proved in that chapter, we can discharge the above assumptions. In this case, each equivalence class will be represented by a single element in the lattice.

Another method by which base lattices can be defined is using the datasort declarations allowed by our sort checker: they allow the declaration of a finite lattice of refinements of a datatype. In this case, the analysis of datasort declarations ensures that the refinements form a lattice.

This approach to base lattices allows refinements of base types to always be represented by base sorts. When we have a refinement like $r_1 \,\&\, r_2$ we represent it by the base sort obtained by applying the intersection operation for the appropriate lattice: $r_1 \overset{a}{\&} r_2$. This leads to simpler algorithms (see Section 3.7) and is the approach used in our sort checker for Standard ML (see Section 8.3).

3.5 Declarative subsorting

The subsorting judgment has the following form, where R and S must be compatible sorts in order for the judgment to be well formed.

$R \leq S$ Sort R is a subsort of S.

It is defined by the following rules which are standard for intersection types (and the same as in the previous chapter) except for the omission of distributivity and the addition of three rules for base sorts: sub_def, sub_inter_def, sub_top_def. The rule sub_def simply embeds the ordering on base sorts into the subsorting judgment. The rules sub_inter_def, sub_top_def ensure that intersections and top sorts in each lattice are accurately reflected by the subsorting judgment.

$$\frac{r \overset{a}{\leq} s}{r \leq s}\,\mathsf{sub\_def} \qquad \frac{}{r^a \,\&\, s^a \leq r^a \overset{a}{\&} s^a}\,\mathsf{sub\_inter\_def} \qquad \frac{}{\top^a \leq \overset{a}{\top}}\,\mathsf{sub\_top\_def}$$

$$\frac{S_1 \leq R_1 \qquad R_2 \leq S_2}{R_1 \to R_2 \leq S_1 \to S_2}\,\mathsf{sub\_arrow} \qquad \frac{}{R \leq R}\,\mathsf{sub\_reflex} \qquad \frac{R_1 \leq R_2 \quad R_2 \leq R_3}{R_1 \leq R_3}\,\mathsf{sub\_trans}$$

$$\frac{}{R \,\&\, S \leq R}\,\mathsf{sub\_inter\_left\_1} \qquad \frac{}{R \,\&\, S \leq S}\,\mathsf{sub\_inter\_left\_2}$$

$$\frac{R \leq S_1 \qquad R \leq S_2}{R \leq S_1 \,\&\, S_2}\,\mathsf{sub\_inter\_right} \qquad \frac{}{R \leq \top^A}\,\mathsf{sub\_top}$$

As in the previous chapter, if $R \leq S$ and $S \leq R$ then we say R and S are *equivalent* sorts, and write $R \cong S$. $\cong$ satisfies the usual properties for an equivalence: it is reflexive (from sub_reflex), transitive (from sub_trans) and symmetric (it has a symmetric definition).

Lemma 3.5.1 ($\cong$ is a Congruence)
If $R_1 \cong S_1$ and $R_2 \cong S_2$ then $R_1 \to R_2 \cong S_1 \to S_2$ and $R_1 \,\&\, R_2 \cong S_1 \,\&\, S_2$.

Proof: Subsorting in each direction can be derived using sub_arrow for the first part, and the rules sub_inter_left_1, sub_inter_left_2 and sub_inter_right for the second part. □

Lemma 3.5.2 (Associativity, Commutativity and Idempotence of &)
$R \,\&\, S \cong S \,\&\, R$ and $R_1 \,\&\, (R_2 \,\&\, R_3) \cong (R_1 \,\&\, R_2) \,\&\, R_3$ and $R \,\&\, R \equiv R$.

Proof: Subsorting in each direction of each part can be derived using only the rules sub_inter_left_1, sub_inter_left_2 and sub_inter_right via sub_reflex. □

Lemma 3.5.3 (Equality of defined and syntactic intersections)

1. $r^a \overset{a}{\&} s^a \cong r^a \,\&\, s^a$
2. $\overset{a}{\top} \cong \top^a$

Proof:

1. Rule sub_inter_def gives one direction, and the other direction is obtained by applying rule sub_inter_right to the two parts of Assumption 3.

2. By sub_top_def and sub_top.

□

3.6 Finiteness of refinements

We will now show that for each type there are only a finite number of refinements up to equivalence. Our proof is quite different from that given by Freeman [FP91], which depends upon the existence of a suitable notion of "splitting" refinements, and is intended to yield a practical algorithm for enumerating the refinements of a type. Instead, we give a simple proof that requires minimal assumptions, and thus should easily extend to other specific instances of refinement types.

First we will need the following definition, which is unambiguous because of Lemma 3.5.2, just like the corresponding definition in the previous chapter. However, in this chapter we work more formally, so we explicitly treat equivalence classes as sets.

Definition 3.6.1
If $\{R_1, \ldots, R_n\}$ is a finite set of sorts each refining type A then $\&\{R_1, \ldots R_n\}$ is defined to be the equivalence class containing $R_1 \,\&\, \ldots \,\&\, R_n$, or $\top^A$ if $n = 0$.

The following lemma captures our intention that the only refinements of base types are base sorts, up to equivalence.

Lemma 3.6.2 (Finiteness of Base Refinements)
For every sort R such that $R \sqsubset a$ for some base type a, there is a base sort s such that $R \cong s$.

Proof: By induction on the structure of R. We have two cases:

Case: $R = s$.
Rule sub_reflex gives $s \leq s$, thus $s \cong s$.

Case: $R = R_1 \,\&\, R_2$.
By the induction hypothesis, we have $R_1 \cong s_1^a$ and $R_2 \cong s_2^a$, i.e., there must be derivations:

$\mathcal{D}_1 :: R_1 \leq s_1^a$ and $\mathcal{D}_1' :: s_1^a \leq R_1$ and $\mathcal{D}_2 :: R_2 \leq s_2^a$ and $\mathcal{D}_2' :: s_2^a \leq R_2$.
We then construct the derivations:

$$\dfrac{\dfrac{\dfrac{}{R_1 \,\&\, R_2 \leq R_1} \quad \dfrac{\mathcal{D}_1}{R_1 \leq s_1^a}}{R_1 \,\&\, R_2 \leq s_1^a} \quad \dfrac{\dfrac{}{R_1 \,\&\, R_2 \leq R_2} \quad \dfrac{\mathcal{D}_2}{R_2 \leq s_2^a}}{R_1 \,\&\, R_2 \leq s_2^a}}{R_1 \,\&\, R_2 \leq s_1^a \,\&\, s_2^a} \quad \dfrac{}{s_1^a \,\&\, s_2^a \leq s_1^{a\,a}\&\, s_2^a}$$
$$R_1 \,\&\, R_2 \leq s_1^{a\,a}\&\, s_2^a$$

and

$$\dfrac{\dfrac{\dfrac{\mathcal{E}_1}{s_1^{a\,a}\&\, s_2^{a\,a} \leq s_1^a}}{s_1^{a\,a}\&\, s_2^a \leq s_1^a} \quad \dfrac{\mathcal{D}_1'}{s_1^a \leq R_1}}{s_1^{a\,a}\&\, s_2^a \leq R_1} \quad \dfrac{\dfrac{\dfrac{\mathcal{E}_2}{s_1^{a\,a}\&\, s_2^{a\,a} \leq s_2^a}}{s_1^{a\,a}\&\, s_2^a \leq s_2^a} \quad \dfrac{\mathcal{D}_2'}{s_2^a \leq R_2}}{s_1^{a\,a}\&\, s_2^a \leq R_2}$$
$$s_1^{a\,a}\&\, s_2^a \leq R_1 \,\&\, R_2$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are the appropriate derivations given by Assumption 3.

□

A simple consequence of this lemma is that each base type has only a finite number of distinct refinements modulo sort equivalence. The following theorem extends this property to all types.

Theorem 3.6.3 (Finiteness of Refinements)
For each type A, there is a finite set Q_A which contains a representative of each of the equivalence classes of refinements of A.

Proof: By induction on the structure of A.

Case: $A = a$.
Then each refinement of a is equivalent to one of the base sorts r^a (by Lemma 3.6.2), and there are only a finite number of these. Thus, we let Q_a be the finite set containing all r^a.

Case: $A = A_1 \to A_2$.
By the induction hypothesis, we have finite sets Q_{A_1}, Q_{A_2}. We define:

$$\begin{aligned} \Psi &= \{R_1 \to R_2 | R_1 \in Q_{A_1}, R_2 \in Q_{A_2}\} \\ Q'_{A_1 \to A_2} &= \{\&\Psi' | \Psi' \subset \Psi\} \end{aligned}$$

We then let $Q_{A_1 \to A_2}$ contain one representative for each element of $Q'_{A_1 \to A_2}$ (each of which is an equivalence class).

Ψ is finite, with size bounded by the product of the sizes of Q_{A_1} and Q_{A_2}. Thus, $Q_{A_1 \to A_2}$ is finite with size bounded by $2^{(\text{size of } \Psi)}$.

It remains to show that every refinement R with $R \sqsubset A_1 \to A_2$ is in one of the equivalence classes in $Q'_{A_1 \to A_2}$. We show this by a nested induction on R. We have three subcases:

Subcase: $R = R_1 \to R_2$.

By inversion, $R_1 \sqsubset A_1$ and $R_2 \sqsubset A_2$.

Thus, R_1 and R_2 are equivalent to some $R'_1 \in Q_{A_1}$ and $R'_2 \in Q_{A_2}$ (using the first induction hypothesis).

Then $R_1 \to R_2 \cong R'_1 \to R'_2$ (by Lemma 3.5.1),
and $R'_1 \to R'_2$ is in equivalence class $\&\{R'_1 \to R'_2\} \in Q'_{A_1 \to A_2}$ (by the above definition of $Q'_{A_1 \to A_2}$).

Subcase: $R = R_1 \,\&\, R_2$.

By inversion, $R_1 \sqsubset A_1 \to A_2$ and $R_2 \sqsubset A_1 \to A_2$.

Thus by the second induction hypothesis, R_1 and R_2 are in equivalence classes $\&\Psi_1$ and $\&\Psi_2$ for some $\Psi_1 \subset \Psi$ and $\Psi_2 \subset \Psi$.

So, $R_1 \& R_2$ is in equivalence class $\&(\Psi_1 \cup \Psi_2)$ (using Lemma 3.5.1 and Lemma 3.5.2), and $\&(\Psi_1 \cup \Psi_2) \in Q'_{A_1 \to A_2}$ (by the definition of $Q'_{A_1 \to A_2}$).

Subcase: $R = \top^{A_1 \to A_2}$.

Using the above definition of $Q'_{A_1 \to A_2}$ and the definition of $\&\Psi'$ when Ψ' is the empty set.

□

While the sets of equivalence classes of refinements constructed in this proof are finite, they can be huge, even for quite small types. This is particularly true as we move towards higher-order types: each increment in the order of the type adds to a stack of exponentials. Since real programs do use higher-order types, any algorithm which requires enumerating equivalence classes of refinements is unlikely to be practical.

3.7 Algorithmic subsorting

The rules in Section 3.5 do not immediately yield an algorithm for deciding subsorting. We thus present an algorithmic subsorting judgment, and show that it is equivalent to the declarative subsorting rules. Due to the absence of distributivity, our subsorting algorithm is quite different from that in the previous chapter, and from previous algorithms for intersection types [Rey88, Fre94].

For efficiency, our algorithm first simplifies the sorts that we wish to compare so that they do not contain any intersections of base sorts. This reflects what is done in our implementation, which represents sorts in simplified form, and simplifies any sorts appearing syntactically in a program. A simplified sort must match the following grammar.

$$\begin{array}{rrl} \text{Simplified Sorts} & R^{\mathsf{s}} & ::= r^a \mid R^{\mathsf{f}} \\ \text{Simplified Function Sorts} & R^{\mathsf{f}} & ::= R_1^{\mathsf{s}} \to R_2^{\mathsf{s}} \mid R_1^{\mathsf{f}} \,\&\, R_2^{\mathsf{f}} \mid \top^{A \to B} \end{array}$$

The following function simplifies a well-formed sort R.

$$\begin{array}{rcll} |r| & = & r & \\ |R \to S| & = & |R| \to |S| & \\ |R \,\&\, S| & = & |R| \overset{a}{\&} |S| & (\text{if } R, S \sqsubset a) \\ |R \,\&\, S| & = & |R| \,\&\, |S| & (\text{if } R, S \sqsubset A \to B) \\ |\top^a| & = & \overset{a}{\top} & \\ |\top^{A \to B}| & = & \top^{A \to B} & \end{array}$$

We now show that this function correctly produces a simplified sort that is equivalent to the input it is given.

Lemma 3.7.1 (Correctness of simplification)
$|R| \cong R$ and $|R|$ is a simplified sort.

Proof: By induction on R.

Case: $R = r$.
Then $|r| = r$ which is a simplified sort satisfying $r \cong r$ (by reflexivity of $\cong$).

Case: $R = R_1 \to R_2$.
By the induction hypothesis on R_1 and R_2, $|R_1| \cong R_1$ and $|R_2| \cong R_2$ and $|R_1|$ and $|R_2|$ are simplified sorts. The result follows by Lemma 3.5.1 and the definition of simplified sorts.

Case: $R = R_1 \,\&\, R_2$, with $R_1, R_2 \sqsubset a$.
By the induction hypothesis $|R_1| \cong R_1$ and $|R_2| \cong R_2$. For the first part we use Lemma 3.5.1, and then Lemma 3.5.3. For the second part, $|R_1|, |R_2|$ must have the forms r_1^a, r_2^a and then $r_1^a \overset{a}{\&} r_2^a$ is defined, and has the form s^a.

Case: $R = R_1 \,\&\, R_2$, with $R_1, R_2 \sqsubset A \to B$.
By the induction hypothesis on R_1 and R_2. We use Lemma 3.5.1 for the first part. For the second part, the induction hypothesis implies that $|R_1|$ and $|R_2|$ are simplified sorts, and that they refine the same type $A \to B$ as R_1 and R_2, so we can use inversion to determine that $|R_1|, |R_2|$ must be simplified function sorts.

Case: $R = \top^a$. Then $|\top^a| = \overset{a}{\top}$, and $\top^a \cong \overset{a}{\top}$ by sub_top_def and sub_top.

Case: $R = \top^{A \to B}$. Then $|\top^{A \to B}| = \top^{A \to B}$. The result follows by sub_reflex and the definition of simplified sorts.

□

We present the core of our algorithm as the following judgment which relates two compatible simplified sorts.

$R^{\mathsf{s}} \trianglelefteq S^{\mathsf{s}}$ $\quad R^{\mathsf{s}}$ determined to be a subsort of S^{s}.

This judgment is defined by the following rules. Here we omit the superscript s to avoid clutter, since every sort in the algorithm is a simplified sort. Later we will sometimes also omit the superscript f when it is clear that a simplified sort refines a function type.

$$\frac{r \overset{a}{\leq} s}{r \trianglelefteq s}\,\mathsf{subalg\_base} \qquad \frac{S_1 \trianglelefteq R_1 \qquad R_2 \trianglelefteq S_2}{R_1 \to R_2 \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_arrow}$$

$$\frac{R_1^{\mathsf{f}} \trianglelefteq S_1 \to S_2}{R_1^{\mathsf{f}} \,\&\, R_2^{\mathsf{f}} \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_\&L1} \qquad \frac{R_2^{\mathsf{f}} \trianglelefteq S_1 \to S_2}{R_1^{\mathsf{f}} \,\&\, R_2^{\mathsf{f}} \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_\&L2}$$

$$\frac{R^{\mathsf{f}} \trianglelefteq S_1^{\mathsf{f}} \qquad R^{\mathsf{f}} \trianglelefteq S_2^{\mathsf{f}}}{R^{\mathsf{f}} \trianglelefteq S_1^{\mathsf{f}} \,\&\, S_2^{\mathsf{f}}}\,\mathsf{subalg\_\&R} \qquad \frac{}{R^{\mathsf{f}} \trianglelefteq \top^{A \to B}}\,\mathsf{subalg\_topR}$$

For base sorts, our algorithm always refers to the partial order of the relevant lattice. For refinements of function sorts, it first breaks down the sort on the right, then the one on the left, and finally uses the standard covariant-contravariant rule. This is somewhat simpler than the corresponding algorithm in Chapter 2 which required an additional judgment in order to account for uses for distributivity.

We now prove some simple lemmas needed to show that algorithmic and declarative subtyping coincide. These are similar to the lemmas required in the previous chapter, although here they are restricted to simplified sorts, and the "algorithmic subsorting for intersections" lemma is only required for simplified function sorts. We show the proofs in detail here. The key proof of transitivity is notably simpler than that in the previous chapter: it can be proved by a structural induction, while in the previous chapter it required induction on the sum of the sizes of the sorts.

Lemma 3.7.2 (Algorithmic subsorting for intersections)
If $R_1^{\mathsf{f}} \trianglelefteq S^{\mathsf{f}}$ *then* $R_1^{\mathsf{f}} \,\&\, R_2^{\mathsf{f}} \trianglelefteq S^{\mathsf{f}}$ *and* $R_2^{\mathsf{f}} \,\&\, R_1^{\mathsf{f}} \trianglelefteq S^{\mathsf{f}}$.

Proof: By induction on S^{f}.

Case: $S^{\mathsf{f}} = S_1^{\mathsf{s}} \to S_2^{\mathsf{s}}$.

By $\mathsf{subalg\_\&L1}$ (first part) and $\mathsf{subalg\_\&L2}$ (second part).

Case: $S^{\mathsf{f}} = S_1^{\mathsf{f}} \,\&\, S_2^{\mathsf{f}}$.

Let $\mathcal{D} :: R_1^{\mathsf{f}} \trianglelefteq S_1^{\mathsf{f}} \,\&\, S_2^{\mathsf{f}}$ be the given derivation.

By inversion, $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ R^{\mathsf{f}} \trianglelefteq S_1^{\mathsf{f}} & R \trianglelefteq S_2^{\mathsf{f}}\end{array}}{R^{\mathsf{f}} \trianglelefteq S_1^{\mathsf{f}} \,\&\, S_2^{\mathsf{f}}}\,\mathsf{subalg\_\&R}$

Applying the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ and then using rule $\mathsf{subalg\_\&R}$ yields the required result.

Case: $S^{\mathsf{f}} = \top^{A \to B}$.

Both parts are immediate, using rule subalg_topR.

□

Lemma 3.7.3 (Reflexivity of Algorithmic Subsorting)
$R^{\mathsf{s}} \trianglelefteq R^{\mathsf{s}}$.

Proof: By induction on R^{s}.

Case: $R^{\mathsf{s}} = r^a$.

By Assumption 1 and rule subalg_base.

Case: $R^{\mathsf{s}} = R_1^{\mathsf{s}} \to R_2^{\mathsf{s}}$.

The induction hypothesis yields $R_1^{\mathsf{s}} \trianglelefteq R_1^{\mathsf{s}}$ and $R_2^{\mathsf{s}} \trianglelefteq R_2^{\mathsf{s}}$. Applying rule subalg_arrow yields the required result.

Case: $R^{\mathsf{s}} = R_1^{\mathsf{f}} \,\&\, R_2^{\mathsf{f}}$.

Using the induction hypothesis on R_1^{f} and R_2^{f}, then Lemma 3.7.2 on each part, then rule subalg_&R.

Case: $R^{\mathsf{s}} = \top^{A \to B}$.

By rule subalg_topR.

□

Lemma 3.7.4 (Transitivity of Algorithmic Subsorting)
If $R_1^{\mathsf{s}} \trianglelefteq R_2^{\mathsf{s}}$ *and* $R_2^{\mathsf{s}} \trianglelefteq R_3^{\mathsf{s}}$ *then* $R_1^{\mathsf{s}} \trianglelefteq R_3^{\mathsf{s}}$.

Proof:
By induction on R_2^{s} and the derivations $\mathcal{D}_2 :: R_2^{\mathsf{s}} \trianglelefteq R_3^{\mathsf{s}}$ and $\mathcal{D}_1 :: R_1^{\mathsf{s}} \trianglelefteq R_2^{\mathsf{s}}$, ordered lexicographically. R_2^{s} is included because some uses of the induction hypothesis swap $\mathcal{D}_1$ and $\mathcal{D}_2$, but in these cases R_2^{s} is decreased (and is not affected by the swap). We have the following cases for $\mathcal{D}_2$.

Case: $\mathcal{D}_2 = \dfrac{r_2 \overset{a}{\leq} r_3}{r_2 \trianglelefteq r_3}\,\mathsf{subalg\_base}$.

Then $\mathcal{D}_1$ must have the form $\dfrac{r_1 \overset{a}{\leq} r_2}{r_1 \trianglelefteq r_2}\,\mathsf{subalg\_base}$

and then we use transitivity of $\overset{a}{\leq}$ (Assumption 2).

Case: $\mathcal{D}_2 = \dfrac{\begin{array}{cc}\mathcal{D}_{21} & \mathcal{D}_{22} \\ R_2 \trianglelefteq S_1 & R_2 \trianglelefteq S_2\end{array}}{R_2 \trianglelefteq S_1 \,\&\, S_2}\,\mathsf{subalg\_\&R}$.

Applying the induction hypothesis to the pairs $\mathcal{D}_1, \mathcal{D}_{21}$ and $\mathcal{D}_1, \mathcal{D}_{22}$ yields the derivations $\mathcal{D}_{31} :: R_1 \trianglelefteq S_1$ and $\mathcal{D}_{32} :: R_1 \trianglelefteq S_2$, to which we apply the rule subalg_&R.

Case: $\mathcal{D}_2 = \dfrac{\begin{array}{c}\mathcal{D}_2' \\ R_{21} \trianglelefteq S_1 \to S_2\end{array}}{R_{21} \,\&\, R_{22} \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_\&L1}$

Then $\mathcal{D}_1$ has the form $\dfrac{\begin{array}{c}\mathcal{D}_{11} \\ R_1 \trianglelefteq R_{21}\end{array} \qquad \begin{array}{c}\mathcal{D}_{12} \\ R_1 \trianglelefteq R_{22}\end{array}}{R_1 \trianglelefteq R_{21} \,\&\, R_{22}}\,\mathsf{subalg\_\&R}.$

Applying the induction hypothesis to $\mathcal{D}_{11}$ and $\mathcal{D}_2'$ yields the required result.

Case: $\mathcal{D}_2 = \dfrac{\begin{array}{c}\mathcal{D}_2' \\ R_{22} \trianglelefteq S_1 \to S_2\end{array}}{R_{21} \,\&\, R_{22} \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_\&L2}.$

Symmetric to the previous case.

Case: $\mathcal{D}_2 = \dfrac{}{R_2 \trianglelefteq \top^{A \to B}}\,\mathsf{subalg\_topR}.$

Immediate, using rule subalg_topR.

Case: $\mathcal{D}_2 = \dfrac{\begin{array}{c}\mathcal{D}_{21} \\ R_{31} \trianglelefteq R_{21}\end{array} \qquad \begin{array}{c}\mathcal{D}_{22} \\ R_{22} \trianglelefteq R_{32}\end{array}}{R_{21} \to R_{22} \trianglelefteq R_{31} \to R_{32}}\,\mathsf{subalg\_arrow}.$

We have the following subcases for $\mathcal{D}_1$:

Subcase: $\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_1' \\ R_{11} \trianglelefteq R_{21} \to R_{22}\end{array}}{R_{11} \,\&\, R_{12} \trianglelefteq R_{21} \to R_{22}}\,\mathsf{subalg\_\&L1}.$

Applying the induction hypothesis to $\mathcal{D}_1'$ and $\mathcal{D}_2$ yields a derivation of $R_{11} \trianglelefteq R_{31} \to R_{32}$, to which we apply rule subalg_&L1.

Subcase: $\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_1' \\ R_{12} \trianglelefteq R_{21} \to R_{22}\end{array}}{R_{11} \,\&\, R_{12} \trianglelefteq R_{21} \to R_{22}}\,\mathsf{subalg\_\&L2}.$

Symmetric to the previous subcase.

Subcase: $\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_{11} \\ R_{21} \trianglelefteq R_{11}\end{array} \qquad \begin{array}{c}\mathcal{D}_{12} \\ R_{12} \trianglelefteq R_{22}\end{array}}{R_{11} \to R_{12} \trianglelefteq R_{21} \to R_{22}}\,\mathsf{subalg\_arrow}.$

By the induction hypothesis on $R_{21}, \mathcal{D}_{21}, \mathcal{D}_{11}$ and $R_{22}, \mathcal{D}_{12}, \mathcal{D}_{22}$ we have $R_{31} \trianglelefteq R_{11}$ and $R_{12} \trianglelefteq R_{32}$. Applying rule subalg_arrow yields the required result.

□

We now make use of the above lemmas to prove that the core of our subsorting algorithm is correct, i.e. that it coincides with the declarative formulation of subsorting on simplified sorts.

Theorem 3.7.5 (Correctness of $\trianglelefteq$) *$|R| \trianglelefteq |S|$ if and only if $R \leq S$.*

Proof:

"Only if" part:

We show the "only if" part by showing that if there is a derivation $\mathcal{D} :: R^{\mathsf{s}} \trianglelefteq S^{\mathsf{s}}$ then there is a derivation $\mathcal{E} :: R^{\mathsf{s}} \leq S^{\mathsf{s}}$, by induction on $\mathcal{D}$.

The result follows by instantiating with $R^{\mathsf{s}} = |R|$ and $S^{\mathsf{s}} = |S|$ to obtain $|R| \leq |S|$, and then using the Correctness of simplification (Lemma 3.7.1) via the transitivity rule to obtain $R \leq S$.

We have the following cases for $\mathcal{D}$.

Case: $\mathcal{D} = \dfrac{r \overset{a}{\leq} s}{r \trianglelefteq s}\,\mathsf{subalg\_base}.$ Then $\mathcal{E} = \dfrac{r \overset{a}{\leq} s}{r \leq s}\,\mathsf{sub\_def}.$

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ S_1 \trianglelefteq R_1 & R_2 \trianglelefteq S_2\end{array}}{R_1 \to R_2 \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_arrow}.$

Then $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ S_1 \leq R_1 & R_2 \leq S_2\end{array}}{R_1 \to R_2 \leq S_1 \to S_2}\,\mathsf{sub\_arrow}$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are obtained by applying the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ R_1 \trianglelefteq S_1 \to S_2\end{array}}{R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_\&L1}.$

We apply the induction hypothesis to $\mathcal{D}_1$ to yield $\mathcal{E}_1$ and then construct the following derivation.

$$\dfrac{\dfrac{}{R_1 \,\&\, R_2 \trianglelefteq R_1}\,\mathsf{sub\_inter\_left\_1} \qquad \begin{array}{c}\mathcal{E}_1 \\ R_1 \leq S_1 \to S_2\end{array}}{R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}\,\mathsf{sub\_trans}.$$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ R_2 \trianglelefteq S_1 \to S_2\end{array}}{R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_\&L2}.$ Dual to the previous case.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ R \trianglelefteq S_1 & R \trianglelefteq S_2\end{array}}{R \trianglelefteq S_1 \,\&\, S_2}\,\mathsf{subalg\_\&R}.$

We apply the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ and then use rule $\mathsf{sub\_inter\_right}$.

Case: $\mathcal{D} = \dfrac{}{R \trianglelefteq \top^{A \to B}}\,\mathsf{subalg\_topR}.$

By rule sub_top.

"If" part:

We now show the "if" part of the theorem by showing that if there is a derivation $\mathcal{E} :: R \leq S$ then there is a derivation $\mathcal{D} :: |R| \trianglelefteq |S|$, by induction on $\mathcal{E}$.

We have the following cases for $\mathcal{E}$.

Case: $\mathcal{E} = \dfrac{r \overset{a}{\leq} s}{r \leq s}\,\mathsf{sub\_def}.$ Then $\mathcal{D} = \dfrac{r \overset{a}{\leq} s}{r \trianglelefteq s}\,\mathsf{subalg\_base}.$

Case: $\mathcal{E} = \dfrac{}{r \,\&\, s \leq r \overset{a}{\&} s}\,\mathsf{sub\_inter\_def}.$

Then $|r \,\&\, s| = r \overset{a}{\&} s$ and $|r \overset{a}{\&} s| = r \overset{a}{\&} s$,
and $r \overset{a}{\&} s \trianglelefteq r \overset{a}{\&} s$ by Lemma 3.7.3 (Reflexivity of $\trianglelefteq$).

Case: $\mathcal{E} = \dfrac{}{R \leq R}\,\mathsf{sub\_reflex}.$ By Lemma 3.7.3 (Reflexivity of $\trianglelefteq$).

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ R_1 \leq R_2 & R_2 \leq R_3\end{array}}{R_1 \leq R_3}\,\mathsf{sub\_trans}.$

By applying the induction hypothesis to $\mathcal{E}_1$ and $\mathcal{E}_2$ and then using Lemma 3.7.4 (Transitivity of $\trianglelefteq$).

Case: $\mathcal{E} = \dfrac{}{R_1 \,\&\, R_2 \leq R_1}\,\mathsf{sub\_inter\_left\_1}$ with $R_1, R_2 \sqsubset a$.

Then $|R_1 \,\&\, R_2| = |R_1| \overset{a}{\&} |R_2|$
and $|R_1| \overset{a}{\&} |R_2| \overset{a}{\leq} |R_1|$ by Assumption 3 ($\overset{a}{\&}$ is lower bound).
Thus, $|R_1| \overset{a}{\&} |R_2| \trianglelefteq |R_1|$ by rule subalg_base.

Case: $\mathcal{E} = \dfrac{}{R_1 \,\&\, R_2 \leq R_1}\,\mathsf{sub\_inter\_left\_1}$ with $R_1, R_2 \sqsubset A \to B$.

Then $|R_1 \,\&\, R_2| \;=\; |R_1| \,\&\, |R_2|$
and $|R_1| \trianglelefteq |R_1|$ by Lemma 3.7.3 (Reflexivity of $\trianglelefteq$)
thus $|R_1| \,\&\, |R_2| \trianglelefteq |R_1|$ by Lemma 3.7.2 (Monotonicity of $\trianglelefteq$).

Case: $\mathcal{E} = \dfrac{}{R_1 \,\&\, R_2 \leq R_2}\,\mathsf{sub\_inter\_left\_2}.$

Symmetric to the previous two cases.

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ R \leq S_1 & R \leq S_2\end{array}}{R \leq S_1 \,\&\, S_2}\,\mathsf{sub\_inter\_right}$ with $R_1, R_2 \sqsubset a$.

Then $|S_1 \,\&\, S_2| = |S_1| \overset{a}{\&} |S_2|$
and $|R| \overset{a}{\leq} |S_1|$ and $|R| \overset{a}{\leq} |S_2|$ by the induction hypothesis.
Thus $|R| \overset{a}{\leq} |S_1| \overset{a}{\&} |S_2|$ by Assumption 4 ($\overset{a}{\&}$ is maximal).

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ R \leq S_1 & R \leq S_2\end{array}}{R \leq S_1 \,\&\, S_2}\,\mathsf{sub\_inter\_right}$ with $R \sqsubset A \to B$.

Then $|S_1 \,\&\, S_2| = |S_1| \,\&\, |S_2|$ and we apply the induction hypothesis to $\mathcal{E}_1$ and $\mathcal{E}_2$ and then apply rule subalg_&R.

Case: $\mathcal{E} = \dfrac{}{R \leq \top^a}\,\mathsf{sub\_top}$ with $R \sqsubset a$.

Then $|\top^a| = \overset{a}{\top}$, and $|R| \overset{a}{\leq} \overset{a}{\top}$ by Assumption 5 ($\overset{a}{\top}$ is maximal).
Thus $|R| \trianglelefteq \overset{a}{\top}$ by rule subalg_base.

Case: $\mathcal{E} = \dfrac{}{R \leq \top^A}\,\mathsf{sub\_top}$ with $R \sqsubset A \to B$.

By rule subalg_topR.

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ S_1 \leq R_1 & R_2 \leq S_2\end{array}}{R_1 \to R_2 \leq S_1 \to S_2}\,\mathsf{sub\_arrow.}$

Then $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ S_1 \trianglelefteq R_1 & R_2 \trianglelefteq S_2\end{array}}{R_1 \to R_2 \trianglelefteq S_1 \to S_2}\,\mathsf{subalg\_arrow}$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are obtained by applying the induction hypothesis to $\mathcal{E}_1$ and $\mathcal{E}_2$.

□

As well as demonstrating the correctness of our algorithm for determining subsorting, we will also make use of this theorem in later proofs to convert between the declarative and algorithmic forms of subsorting. In particular, we will often convert to the algorithmic form to reduce the number of cases that we need to consider.

3.8 Declarative sort assignment

We now present the sort assignment judgment for terms. As in the previous chapter, this judgment requires a term M which satisfies $\Gamma \vdash_\Sigma M{:}A$, $\Delta \sqsubset \Gamma$ and $R \sqsubset A$ in order to be well formed.

$\Delta \vdash_\Sigma M \in R$ Term M has sort R under Δ and Σ.

The sort assignment rules are standard, and the same as those in the previous chapter, except that the introduction rules for intersection and top are restricted to values. The signature Σ is fixed throughout these rules, and we often omit it to avoid clutter (here and in the remainder of this chapter).

$$\frac{x{\in}R \text{ in } \Delta}{\Delta \vdash x \in R}\,\mathsf{sa\_var} \qquad \frac{c{\in}R \text{ in } \Sigma}{\Delta \vdash_\Sigma c \in R}\,\mathsf{sa\_const}$$

$$\frac{\Delta, x{\in}R \vdash M \in S}{\Delta \vdash \lambda x{:}A.M \in R \to S}\,\mathsf{sa\_lam} \qquad \frac{\Delta \vdash M \in R \to S \quad \Delta \vdash N \in R}{\Delta \vdash M\,N \in S}\,\mathsf{sa\_app}$$

$$\frac{\Delta \vdash V \in R \qquad \Delta \vdash V \in S}{\Delta \vdash V \in R \,\&\, S}\,\mathsf{sa\_inter} \qquad \frac{}{\Delta \vdash V \in \top^A}\,\mathsf{sa\_top}$$

$$\frac{\Delta \vdash M \in R \qquad R \leq S}{\Delta \vdash M \in S}\,\mathsf{sa\_subs}$$

In order to demonstrate that these rules are sensible we demonstrate that they satisfy some standard properties. The most important is subject reduction, i.e. that reduction preserves sorts. But first we need to show some structural properties: weakening, exchange, contraction and substitution. These are standard properties for hypothetical judgments (see e.g. [Pfe05]): in this case the hypotheses are the assumed sorts for variables in the sort context.

Lemma 3.8.1 (Weakening, Exchange, Contraction)

1. *If* $\Delta \vdash M \in R$ *then* $\Delta, x{\in}S \vdash M \in R$.
2. *If* $\Delta, x{\in}S_1, y{\in}S_2, \Delta' \vdash M \in R$ *then* $\Delta, y{\in}S_2, x{\in}S_1, \Delta' \vdash M \in R$.
3. *If* $\Delta, x{\in}S, y{\in}S, \Delta' \vdash M \in R$ *then* $\Delta, w{\in}S, \Delta' \vdash \{w/x\}\{w/y\}M \in R$.

Proof: In each case by induction over the structure of the given sorting derivation. The cases for the rule sa_var are straightforward; the cases for other rules simply follow the structure of the given derivation. □

The following lemma demonstrates that values are preserved by substitution of values for variables.

Lemma 3.8.2 (Value Preservation)

1. $\{V_1/x\}V^{\mathsf{a}}$ *is an atom.*
2. $\{V_1/x\}V$ *is a value.*

Proof: By a straightforward structural induction on V^{a} and V. □

We are now in a position to prove the substitution lemma. We implicitly rely on exchange, both in the statement of the lemma and in its proof.

Lemma 3.8.3 (Substitution Lemma)
If $\Delta \vdash V \in R$ *and* $\Delta, x{\in}R \vdash N \in S$ *then* $\Delta \vdash \{V/x\}N \in S$.

Proof:
Let $\mathcal{D}_1 :: \Delta \vdash V \in R$ and $\mathcal{D}_2 :: \Delta, x{\in}R \vdash N \in S$ be the given derivations. The proof is by a simple induction on $\mathcal{D}_2$, constructing the derivation $\mathcal{D}_3 :: \Delta \vdash \{V/x\}N \in S$. We show three interesting cases.

Case: $\mathcal{D}_2 = \dfrac{x{\in}R \text{ in } (\Delta, x{\in}R)}{\Delta, x{\in}R \vdash x \in R}\,\mathsf{sa\_var}.$

Then $R = S$ and $N = x$ thus $\{V/x\}N = V$.
So we can simply choose $\mathcal{D}_3 = \mathcal{D}_1 :: \Delta \vdash V \in R$.

Case: $\mathcal{D}_2 = \dfrac{y{\in}S \text{ in } (\Delta, x{\in}R)}{\Delta, x{\in}R \vdash y \in S}\,\mathsf{sa\_var}$ with $y \neq x$.

Then $N = y$ and $\{V/x\}y = y$ and $y{\in}S$ is in Δ so we can simply use the variable rule:

$$\mathcal{D}_3 = \frac{y{\in}S \text{ in } \Delta}{\Delta \vdash y \in S}\,\mathsf{sa\_var}.$$

Case: $\mathcal{D}_2 = \dfrac{\begin{array}{cc}\mathcal{D}_{21} & \mathcal{D}_{22} \\ \Delta, x{\in}R \vdash V' \in S_1 & \Delta, x{\in}R \vdash V' \in S_2\end{array}}{\Delta, x{\in}R \vdash V' \in R \,\&\, S}\,\mathsf{sa\_inter}.$

Applying the induction hypothesis to $\mathcal{D}_{21}$ and $\mathcal{D}_{22}$ yields the derivations $\mathcal{D}_{31} :: \Delta \vdash \{V/x\}V' \in S_1$ and $\mathcal{D}_{32} :: \Delta \vdash \{V/x\}V' \in S_2$.
Since $\{V/x\}V'$ is a value (by the value preservation lemma above) we can apply rule $\mathsf{sa\_inter}$ to these derivations to obtain $\mathcal{D}_3 :: \Delta \vdash \{V/x\}V' \in S_1 \,\&\, S_2$, as required.

The remaining cases simply reconstruct the derivation $\mathcal{D}_3$ following the structure of $\mathcal{D}_2$, similar to the case for $\mathsf{sa\_inter}$ above. □

This proof is somewhat independent of the exact definition of values: the main requirement is that the substitution $\{V/x\}V'$ always yields a value. Thus, a similar result holds for alternative definitions of values that satisfy this criteria. E.g. we could define values to include all terms, so that the "value" restriction in rule sa_inter becomes vacuous. Then the above lemma would be the substitution lemma for a system without distributivity but with no value restriction. Such alternative definitions are not the main focus of this dissertation, but we will occasionally consider them where they are particularly interesting or where we feel that they demonstrate the robustness of our results.

A simple corollary of the Substitution Lemma is obtained by considering the case where V is a variable, as follows.

Corollary 3.8.4 (Variable Subsumption)
If $\Delta, x{\in}R \vdash M \in S$ *and* $R' \leq R$ *then* $\Delta, x{\in}R' \vdash M \in S$.

Proof:

$\Delta, x{\in}R \vdash M \in S$	By assumption
$\Delta, y{\in}R', x{\in}R \vdash M \in S$ (where y is not in $\Delta, x{\in}R$)	By weakening (Lemma 3.8.1 part 1)
$\Delta, y{\in}R' \vdash y \in R'$	By rule sa_var
$R' \leq R$	By assumption
$\Delta, y{\in}R' \vdash y \in R$	By rule sa_subs
$\Delta, y{\in}R' \vdash \{y/x\}M \in S$	By the Substitution Lemma (3.8.3)
$\Delta, x{\in}R' \vdash M \in S$	Replacing y by x (x is not in $\Delta, y{\in}R'$ nor in $\{y/x\}M$)

□

(It appears that this result also holds with alternative definitions of values that exclude variables, although then it must be proved separately since the substitution lemma only applies to values.)

The following property is critical to our proof of subject reduction. It generalizes similar properties in languages without subtyping or intersections.

Lemma 3.8.5 (Inversion for λ-Abstractions)
If $\Delta \vdash \lambda x{:}A_1.M \in R$ *and* $R \leq S_1 \to S_2$ *then* $\Delta, x{\in}S_1 \vdash M \in S_2$.

Proof: By induction on the derivation of the first assumption, $\mathcal{D} :: \Delta \vdash \lambda x{:}A_1.M \in R$. There are three possible cases for $\mathcal{D}$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Delta, x{\in}R_1 \vdash M \in R_2\end{array}}{\Delta \vdash \lambda x{:}A.M \in R_1 \to R_2}\,\mathsf{sa\_lam}.$

Then $R = R_1 \to R_2$ and so $R_1 \to R_2 \leq S_1 \to S_2$ (by assumption).
But then there must be a derivation $\mathcal{E} :: |R_1| \to |R_2| \trianglelefteq |S_1| \to |S_2|$ (by Correctness of Algorithmic Subsorting, Theorem 3.7.5).

By inversion $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ |S_1| \trianglelefteq |R_1| & |R_2| \trianglelefteq |S_2|\end{array}}{|R_1| \to |R_2| \trianglelefteq |S_1| \to |S_2|}\,\mathsf{subalg\_arrow}.$

Thus $S_1 \leq R_1$ and $R_2 \leq S_2$ (also by Correctness of Algorithmic Subsorting).
So $\Delta, x{\in}R_1 \vdash M \in S_2$ (by rule sa_subs)
and finally $\Delta, x{\in}S_1 \vdash M \in S_2$ (by Variable Subsumption, Corollary 3.8.4).

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E} \\ \Delta \vdash M \in R' & R' \leq R\end{array}}{\Delta \vdash M \in R}\,\mathsf{sa\_subs}.$

Then $R' \leq S_1 \to S_2$ (By rule sub_trans), so we can apply the induction hypothesis to D_1 to obtain $\Delta, x{\in}S_1 \vdash M \in S_2$, as required.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash V \in R_1 & \Delta \vdash V \in R_2\end{array}}{\Delta \vdash V \in R_1 \,\&\, R_2}\,\mathsf{sa\_inter}.$

Then $R = R_1 \,\&\, R_2$ and so $R_1 \,\&\, R_2 \leq S_1 \to S_2$ (by assumption).
Applying inversion to the corresponding algorithmic subsorting derivation (via the Correctness of Algorithmic Subsorting in the same way as the case for rule sa_lam) we find that we must have one of the following subcases:

Subcase: $R_1 \leq S_1 \to S_2$ (corresponding to rule subalg_&L1).
Then we apply the induction hypothesis to D_1, yielding $\Delta, x{\in}S_1 \vdash M \in S_2$, as required.

Subcase: $R_2 \leq S_1 \to S_2$ (corresponding to rule subalg_&L2).
Symmetric to the previous subcase.

□

The last case of this proof depends upon the strong inversion properties obtained from the algorithmic form of subsorting, in particular that $R_1 \,\&\, R_2 \leq S_1 \to S_2$ only if $R_1 \leq S_1 \to S_2$ or $R_2 \leq S_1 \to S_2$. This property fails if we allow the distributivity rule, resulting in the need for a more complicated approach in Chapter 2.

We are now in a position to prove the main theorem of this subsection.

Theorem 3.8.6 (Sort Subject Reduction)
If $\Delta \vdash M \in R$ *and* $M \mapsto N$ *then* $\Delta \vdash N \in R$.

Proof: By induction on the structure of the derivation $\mathcal{D} :: \Delta \vdash M \in R$. We have the following cases for $\mathcal{D}$ and the derivation $\mathcal{E} :: M \mapsto N$. We first treat the three sorting rules which do not correspond to term constructs, and then consider the remaining cases by inversion on $\mathcal{E}$.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash V \in R_1 & \Delta \vdash V \in R_2\end{array}}{\Delta \vdash V \in R_1 \,\&\, R_2}\,\mathsf{sa\_inter}.$

Then $M = V$ and by inversion M must have the form $V^{\mathsf{a}}\,V$ or $\lambda x{:}A.M'$. We show the

second subcase: the first is similar.
The derivation of $M \mapsto N$ must be by rule reduce_lam, so $N \mapsto N'$ and $N = \lambda x{:}A.N'$.

By the induction hypothesis on we have $\Delta \vdash \lambda x{:}A.N' \in R_1$ and $\Delta \vdash \lambda x{:}A.N' \in R_2$.
We then apply rule sa_inter.

Case: $\mathcal{D} = \dfrac{}{\Delta \vdash V \in \top^A}\,\mathsf{sa\_top}.$

Then $M = V$ and so by inversion M must have the form $V^{\mathsf{a}}\,V$ or $\lambda x{:}A.M'$. We show latter subcase: the former is similar.
As in the previous case, $N = \lambda x{:}A.N'$ which is a value. Thus $\Delta \vdash N \in \top^A$ (by rule sa_top).

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Delta \vdash M \in R_1 \end{array} \qquad R_1 \leq R}{\Delta \vdash M \in R}\,\mathsf{sa\_subs}.$

Then $\Delta \vdash N \in R_1$ (by the induction hypothesis on $\mathcal{D}_1$),
thus $\Delta \vdash N \in R$ (by rule sa_subs).

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_2\\ M \mapsto N\end{array}}{\lambda x{:}A.M \mapsto \lambda x{:}A.N}\,\mathsf{reduce\_lam}$

and

$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2\\ \Delta, x{\in}R_1 \vdash M \in R_2\end{array}}{\Delta \vdash \lambda x{:}A.M \in R_1 \to R_2}\,\mathsf{sa\_lam}.$

By the induction hypothesis on $\mathcal{D}_2, \mathcal{E}_2$ and then using rule sa_lam.

Case: $\mathcal{E}$ is an instance of reduce_app1 or reduce_app2. Similar to the previous case.

Case: $\mathcal{E} = \dfrac{}{(\lambda x{:}A.M_1)\,V \mapsto \{V/x\}M_1}\,\beta_v$

and

$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Delta \vdash \lambda x{:}A.M_1 \in R_2 \to R\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Delta \vdash V \in R_2\end{array}}{\Delta \vdash (\lambda x{:}A.M_1)\,V \in R}\,\mathsf{sa\_app}.$

Then we apply the Inversion Lemma for λ (Lemma 3.8.5) to $\mathcal{D}_1$ to obtain

$$\Delta, x{\in}R_2 \vdash M_1 \in R$$

and then we apply the Substitution Lemma (Lemma 3.8.3), using $\mathcal{D}_2$, to show

$$\Delta \vdash \{V/x\}M_1 \in R$$

as required.

□

This theorem demonstrates that our calculus and sorting rules are at least basically sensible and internally consistent. In Chapter 4 we use similar proof techniques to show a corresponding sort preservation result for a functional language with reference cells.

The above proof is somewhat robust to changes in the definition of values, although it does require that if $V \mapsto N$ then N is a value, which seems likely to be true for most sensible definitions of values. In fact, generally values are defined in such a way that they do not reduce (usually by disallowing reduction within λ-abstractions), in which case this requirement would be vacuously satisfied.

3.9 Sorts for constants

In $\lambda_v^{\to\&}$ we consider that functions may have effects, and so we omit the distributivity rules. However, constants also have function types, but applications of constants can never result in effects. We have thus included repeated applications of constants as a form of value, and thus intersection introduction is allowed for such applications. This leads to a some awkward technical complications, because we can have an expression $V^{\mathsf{a}}\, V \in R_1 \,\&\, R_2$, but with no sort S such that $V \in S$ and $V^{\mathsf{a}} \in S \to (R_1 \,\&\, R_2)$.

To avoid these complications, the validity of signatures requires that the sorts of constants be distributive. This condition itself is somewhat awkward, but it seems the most natural way to avoid the complications. In our implementation this property holds naturally for the sorts generated for constructors by the analysis of datasort declarations: when generating lattice elements for intersections of datasorts, we must consider sorts for constructors which map into that intersection. It also seems likely that the distributivity property would hold in other situations.

In general, any sort can be "completed" into a distributive one: roughly whenever we have $(R_1 \to R_2) \,\&\, (S_1 \to S_2)$ we add a conjunct for $(R_1 \,\&\, S_1) \to (R_2 \,\&\, S_2)$. An alternative to our presentation would be to allow any sort to be declared for a constant in a signature, and to perform this completion in the sort assignment rule for constants. This would avoid the awkward condition in the validity of signatures. We have chosen not to do this because the completion can result in exponentially larger sorts, and it is not an accurate reflection of what we do in our implementation.

Another alternative might be to include a restricted form of the distributivity subsorting rule that is used only for constants: essentially our definition of "distributive" sorts is designed to ensure that it does not matter whether we have distributivity. We could even use a separate type constructor for "pure functions", including constants (as is done by Harper and Stone [HS00] in their type theoretic interpretation of SML), and include distributivity for such functions. We have chosen not to follow this alternative because it leads to some complications in the sort checking algorithm, which do not seem justified given that the sorts for constructors are naturally distributive in the case we are most interested in.

The following lemma verifies that our approach achieves the desired effect. It is critical in the proof of the inversion lemma for applications (Lemma 3.10.2), which itself is critical to the completeness of our sort checking algorithm. We state the lemma in a generalized form suitable for an inductive proof.

Lemma 3.9.1 (Distributivity for atoms)

If $\Delta \vdash V^{\mathsf{a}} \in R$ *and* $R \leq R_1 \to \ldots \to R_n$
and $\Delta \vdash V^{\mathsf{a}} \in S$ *and* $S \leq S_1 \to \ldots \to S_n$
then $\Delta \vdash V^{\mathsf{a}} \in (R_1 \,\&\, S_1) \to \ldots \to (R_n \,\&\, S_n)$.

Proof: By induction on the two sorting derivations, lexicographically. The cases where one derivation or the other uses subsumption or intersection introduction follow directly from the induction hypothesis. The case where both use sa_app requires applying the induction hypothesis with a smaller V^{a}, using intersection introduction to combine two sorts for the argument value. The case where both use sa_const follows directly from the distributivity requirement. □

Lemma 3.9.2 ($\top$ distributivity for atoms)
If $\Gamma \vdash V^{\mathsf{a}} : A_1 \to \ldots \to A_n$ *and* $\Delta \sqsubset \Gamma$
then $\Delta \vdash V^{\mathsf{a}} \in \top^{A_1} \to \ldots \to \top^{A_n}$.

Proof: By induction on the structure of V^{a}. The case for c follows directly from the distributivity requirement. The case for $V^{\mathsf{a}}\, V$ uses the induction hypothesis, followed by the rule sa_app (with V assigned the sort $\top^A$ using sa_top). □

3.10 Principal sorts and decidability of inference

Two important properties of ML are the existence of principal type schemes and a practical algorithm for finding them. We now consider the corresponding properties for the sorts of $\lambda_v^{\to\&}$. We find that principal sorts exist, and that there is an algorithm for finding them, but both results seem to be mostly of theoretical interest. This is because they require enumeration of all refinements of a type, which is not practical in general. The work by Freeman [Fre94] and experience with the implementation described later in this dissertation suggests that the super-exponential bound on the number of equivalence classes in the proof of Theorem 3.6.3 (Finiteness of Refinements) is an accurate reflection the huge number of unique refinements at higher-order types. Regardless, these theorems and proofs are still interesting, if only to demonstrate where the enumeration is required.

The principal sorts theorem applies only to values: other terms do not necessarily have principal sorts. This is consistent with Standard ML, which has principal type schemes for values, but not for all expressions. E.g. the Standard ML expression `ref nil` can be assigned the type `(int list) ref` and also the type `(bool list) ref` but it can not be assigned the generalized type scheme `('a list) ref` (see [MTHM97]).

Theorem 3.10.1 (Principal Sorts)
If $\Gamma \vdash V : A$ *and* $\Delta \sqsubset \Gamma$ *then there is some* R_1 *such that* $R_1 \sqsubset A$ *and* $\Delta \vdash V \in R_1$ *and for all* R_2 *such that* $R_2 \sqsubset A$ *and* $\Delta \vdash V \in R_2$ *we have* $R_1 \leq R_2$.

Proof: We make use of the finite set Q_A from Theorem 3.6.3 which contains representatives of all equivalence classes of refinements of A.

We construct the finite set

$$Q = \{S_1 \in Q_A \mid \Delta \vdash V \in S_1\}$$

and then we choose

$$R_1 = \&Q.$$

This satisfies the first requirement of the theorem: $\Delta \vdash V \in R_1$ (by rule sa_top and repeated use of rule sa_inter following the structure of R_1).

It also satisfies the second requirement of the theorem: if $\Delta \vdash V \in R_2$ then R_2 is equivalent to some $S_2 \in Q$ and so $R_1 \leq R_2$ (by transitivity and repeated use of rules sub_inter_left1 and sub_inter_left2). □

This proof is not constructive: it does not directly yield an algorithm for inferring principal sorts, because it does not specify a method for checking $\Delta \vdash V \in S_1$ for each S_1. We now show that this problem is decidable. Our proof makes use of the finite set of equivalence classes Q_A constructed for each type A in Theorem 3.6.3 (Finiteness of Refinements). These sets can be impractically large, and so the algorithm in our proof is far from practical. Further, the previous proof also uses Q_A which suggests that the principal sorts themselves might be too large to be practical.

We will need the following inversion lemmas.

Lemma 3.10.2 (Inversion for Applications)
$\Delta \vdash M_1\, M_2 \in R$ if and only if there is a sort S such that $\Delta \vdash M_1 \in S \to R$ and $\Delta \vdash M_2 \in S$.

Proof:

"If" part: By rule sa_app.

"Only if" part: By induction on the structure of the assumed derivation $\mathcal{D} :: \Delta \vdash M_1\, M_2 \in R$. We have the following cases.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash M_1 \in R_2 \to R & \Delta \vdash M_2 \in R_2\end{array}}{\Delta \vdash M_1\, M_2 \in R}\,\mathsf{sa\_app}.$

Then we let $S = R_2$, and use $\mathcal{D}_1$ and $\mathcal{D}_2$.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}' & \\ \Delta \vdash M_1\, M_2 \in R' & R' \leq R\end{array}}{\Delta \vdash M_1\, M_2 \in R}\,\mathsf{sa\_subs}.$

Then there exists $S' \leq R'$ such that $\Delta \vdash M_1 \in S' \to R$ and $\Delta \vdash M_2 \in S'$ (by the induction hypothesis on $\mathcal{D}'$). But $S' \leq R$ (by rule sub_trans) and so we have the required result by choosing $S = S'$.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash M_1\, M_2 \in R_1 & \Delta \vdash M_1\, M_2 \in R_2\end{array}}{\Delta \vdash M_1\, M_2 \in R_1 \,\&\, R_2}\,\mathsf{sa\_inter}.$

Then $M_1\, M_2$ is a value, so $M_1\, M_2 = V^{\mathsf{a}}\, V$. Applying the ind. hyp. to $\mathcal{D}_1$ and $\mathcal{D}_2$ yields

S_1 with $\Delta \vdash V^{\mathsf{a}} \in S_1 \to R_1$, $\Delta \vdash V \in S_1$, and
S_2 with $\Delta \vdash V^{\mathsf{a}} \in S_2 \to R_2$, $\Delta \vdash V \in S_2$.
Then Distributivity for Atoms (Lemma 3.9.1) yields
$\Delta \vdash V^{\mathsf{a}} \in (S_1 \,\&\, S_2) \to (R_1 \,\&\, R_2)$ and sa_inter yields
$\Delta \vdash V \in (S_1 \,\&\, S_2)$. Applying sa_app yields the required result.

Case: $\mathcal{D} = \dfrac{}{\Delta \vdash M_1\, M_2 \in \top^A}\,\mathsf{sa\_top}.$

Then $M_1\, M_2 = V^{\mathsf{a}}\, V$. The result follows by Lemma 3.9.1 ($\top$-distributivity for Atoms).

□

Lemma 3.10.3 (Inversion for Variables)
$\Delta \vdash x \in R$ if and only if there is some S such that $x{\in}S$ is in Δ and $S \leq R$.

Proof:

"If" part: We simply construct the derivation

$$\dfrac{\dfrac{x{\in}S \text{ in } \Delta}{\Delta \vdash x \in S}\,\mathsf{sa\_var} \qquad S \leq R}{\Delta \vdash x \in R}\,\mathsf{sa\_subs}.$$

"Only if" part: By induction on the structure of the assumed derivation $\mathcal{D} :: \Delta \vdash x \in R$. We have the following cases.

Case: $\mathcal{D} = \dfrac{x{\in}R \text{ in } \Delta}{\Delta \vdash x \in R}\,\mathsf{sa\_var}.$

Then we choose $S = R$ and $R \leq R$ (by rule sub_reflex).

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Delta \vdash x \in R_1\end{array} \qquad R_1 \leq R}{\Delta \vdash x \in R}\,\mathsf{sa\_subs}.$

Then there is S such that $S \leq R_1$ and $x{\in}S$ in Δ (by the induction hypothesis on $\mathcal{D}_1$), and so $S \leq R$ (by rule sub_trans).

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Delta \vdash x \in R_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Delta \vdash x \in R_2\end{array}}{\Delta \vdash x \in R_1 \,\&\, R_2}\,\mathsf{sa\_inter}.$

Then $R = R_1 \,\&\, R_2$.
Now $S \leq R_1$ and $S \leq R_2$ (induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2$).
Thus $S \leq R_1 \,\&\, R_2$ (by rule sub_inter_right).

Case: $\mathcal{D} = \dfrac{}{\Delta \vdash x \in \top^A}\,\mathsf{sa\_top}.$

Then $R = \top^A$ and there must be some $x{\in}S$ in Δ (by the refinement restriction). Finally, $S \leq \top^A$ (by rule sub_top).

□

Lemma 3.10.4 (Inversion for Constants)
If $c{\in}S$ is in Σ then $\Delta \vdash_\Sigma c \in R$ if and only if $S \leq R$.

Proof: Almost identical to the previous proof. □

The final inversion lemma is a simple corollary of the Inversion Lemma for λ-Abstractions (Lemma 3.8.5).

Corollary 3.10.5 (Reflexive Inversion for λ-Abstractions)
$\Delta \vdash \lambda x{:}A.M_2 \in R_1 \to R_2$ if and only if $\Delta, x{\in}R_1 \vdash M_2 \in R_2$.

Proof:

"If" part:
By rule sa_lam.

"Only if" part:
By the Inversion Lemma for λ-Abstractions (Lemma 3.8.5) applied to $R_1 \to R_2 \leq R_1 \to R_2$ (which is obtained by rule sub_reflex).

□

Theorem 3.10.6 (Decidability of Sorting)
Given $\Gamma \vdash M : A$ and $\Delta \sqsubset \Gamma$ and $R \sqsubset A$ there is a procedure for determining whether there is a derivation of $\Delta \vdash M \in R$.

Proof: By induction on the structure of the derivation $\mathcal{D} :: \Gamma \vdash M : A$ (or equivalently, by induction on the structure of M) and the structure of R lexicographically. We have the following cases.

Case: $\mathcal{D} = \dfrac{x{:}A \text{ in } \Gamma}{\Gamma \vdash x : A}\,\mathsf{tp\_var}.$

Then $M = x$ and since $\Delta \sqsubset \Gamma$ there must also be some $S \sqsubset A$ such that $x{\in}S$ is in Δ.

To determine whether there is a derivation of $\Delta \vdash M \in R$ we simply use the subsorting algorithm in Section 3.7 to check whether $S \leq R$ holds.

By the above Inversion Lemma for Variables (Lemma 3.10.3), and because x appears only once in Δ, this correctly determines whether $\Delta \vdash x \in R$.

Case: $\mathcal{D} = \dfrac{c{:}a \text{ in } \Sigma}{\Gamma \vdash c : a}\,\mathsf{tp\_const}.$

Similar to the previous case.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \vdash M_1 : B \to A & \Gamma \vdash M_2 : B\end{array}}{\Gamma \vdash M_1\, M_2 : A}\,\mathsf{tp\_app}.$

Then $M = M_1\, M_2$.
Let Q_B be the finite set in the proof of the Finiteness of Refinements Theorem (Theorem 3.6.3).

To determine whether $\Delta \vdash M_1\, M_2 \in R$ we try each $S \in Q_B$ and check whether $\Delta \vdash M_1 \in S \to R$ and $\Delta \vdash M_2 \in S$ (which we can determine, by the induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2$).

This correctly determines whether $\Delta \vdash M_1\, M_2 \in R$ holds because this is so if and only if there is a sort S such that $\Delta \vdash M_1 \in S \to R$ and $\Delta \vdash M_2 \in S$ (by the Inversion Lemma for Applications above, 3.10.2), and it suffices to consider only $S \in Q_B$ (by rule $\mathsf{sa\_sub}$ and the Finiteness of Refinements Theorem).

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2 \\ \Gamma, x{:}A \vdash_\Sigma M' : B\end{array}}{\Gamma \vdash \lambda x{:}A.M' : A \to B}\,\mathsf{tp\_lam}.$

Then $M = \lambda x{:}A.M'$. We have three subcases.

Subcase: $R = R_1 \to R_2$.
We simply determine whether $\Delta, x{\in}R_1 \vdash M' \in R_2$, using the procedure obtained by applying the induction hypothesis to $\mathcal{D}_2, R_2$.
By the Reflexive Inversion Corollary for λ-Abstractions (Corollary 3.10.5) this correctly determines $\Delta \vdash \lambda x{:}A.M' \in R_1 \to R_2$, as required.

Subcase: $R = R_1 \,\&\, R_2$.
Then

$$\Delta \vdash \lambda x{:}A.M' \in R_1 \,\&\, R_2$$

iff

$$\Delta \vdash \lambda x{:}A.M' \in R_1 \quad \text{and} \quad \Delta \vdash \lambda x{:}A.M' \in R_2$$

(by rule $\mathsf{sa\_inter}$ in one direction; by rules $\mathsf{sub\_inter\_left\_2}$, $\mathsf{sub\_inter\_left\_2}$ and $\mathsf{sa\_subs}$ in the other).
We thus simply determine whether both $\Delta \vdash \lambda x{:}A.M' \in R_1$ and $\Delta \vdash \lambda x{:}A.M' \in R_2$ hold by using the procedures obtained by applying the induction hypothesis to $\mathcal{D}, R_1$ and $\mathcal{D}, R_2$ respectively.

Subcase: $R = \top^A$.
Then we determine that $\Delta \vdash \lambda x{:}A.M' \in \top^A$ holds, by rule $\mathsf{sa\_top}$.

□

3.11 Bidirectional sort checking

The proofs in the previous section do not lead to practical algorithms for inferring principal sorts for terms nor for deciding whether a term has a particular sort. This is because they rely on enumerating the set of unique refinements of a type, which can be huge. Previous work on refinements for ML by Freeman [Fre94] has attempted to construct a practical algorithm for inferring principal sorts by using program analysis techniques from abstract interpretation to avoid this enumeration as much as possible. This work also efficiently generated the unique refinements by representing each base sort as a union of "unsplittable" components.

This appears to work quite well when inferring sorts for first-order functions, with a small number of base sorts. Alas, for many real ML programs it seems that sort inference is infeasible, in part because the principal types themselves can be huge. These sorts generally reflect many "accidental" properties of programs beyond what the programmer intends. These accidental properties not only complicate sort inference, they also have undesirable consequences when there is an error in a program. For example, an application of a function to an inappropriate argument may be accepted because the argument matches an "accidental" part of the inferred sort for the function.

Our solution to these problems is to move away from sort inference and program analysis techniques, and instead move towards *sort checking* and techniques based on type systems. Type systems for programming languages are generally designed with the assumption that the programmer will have particular types in mind as they write their program. We assume similarly for sorts, since they are essentially a refined level of types designed specifically to express properties. Further, to write correct code a programmer needs to have in mind the intended properties of their code. In this section we show that if a programmer declares a small number of the intended properties using sort annotations then we can check the properties by bidirectional sort checking without enumerating refinements.

Our algorithm has some elements that are similar to the proof of Decidability of Sorting (Theorem 3.10.6). In particular, it checks λ-abstractions against a goal sort in essentially the same way. However for function applications it avoids the enumeration by a syntactic restriction that ensures that the function is only checked against a sort, and its sort need never be inferred. This syntactic restriction seems quite reasonable in practice: for ML programs it generally only requires that the intended sorts be declared at `fun` definitions.

Our syntactic restriction is similar to that in the programming language Forsythe [Rey88, Rey96], which also includes intersections, and uses a bidirectional type checking algorithm. Our restriction is somewhat simpler though: we only distinguish two classes of terms, while Forsythe allows more terms but requires an countably infinite set of syntactic classes. Our algorithm for sort checking is very different from that for Forsythe, in part due to our value restriction, but also because the Forsythe algorithm does not extend to arbitrary base lattices (see Section 2.9).

3.11.1 Syntax

As in the previous chapter, the bidirectional checking algorithm requires a new term constructor for sort annotations with a list of alternative goal sorts, and uses the following two syntactic subclasses of terms.

$$\begin{array}{rrl} \text{Inferable Terms} & I & ::= c \mid x \mid I\,C \mid (C{\in}L) \\ \text{Checkable Terms} & C & ::= I \mid \lambda x{:}A.C \\ \text{Sort Constraints} & L & ::= \cdot \mid L, R \end{array}$$

3.11.2 Sort checking algorithm

The sorting judgments for inferable and checkable terms are defined for terms $M = I$ or C such that $\Gamma \vdash M : A$ and $\Delta \sqsubset \Gamma$ and $R \sqsubset A$.

$\Delta \vdash I \overset{\Rightarrow}{\in} R$ Term I has R as an inferable sort.

$\Delta \vdash C \overset{\Leftarrow}{\in} R$ Term C checks against sort R.

It is our intention that the rules for these judgments be interpreted algorithmically as follows.

1. Given Δ, Σ and I, we can synthesize each R such that there is a derivation of $\Delta \vdash I \overset{\Rightarrow}{\in} R$.
2. Given Δ, Σ, C, and R we can check whether there is a derivation of $\Delta \vdash C \overset{\Leftarrow}{\in} R$.

$$\frac{x{\in}R \text{ in } \Delta}{\Delta \vdash x \overset{\Rightarrow}{\in} R}\,\mathsf{si\_var} \qquad \frac{c{\in}R \text{ in } \Sigma}{\Delta \vdash c \overset{\Rightarrow}{\in} R}\,\mathsf{si\_const} \qquad \frac{\Delta \vdash I \overset{\Rightarrow}{\in} R_1 \to R_2 \qquad \Delta \vdash C \overset{\Leftarrow}{\in} R_1}{\Delta \vdash I\,C \overset{\Rightarrow}{\in} R_2}\,\mathsf{si\_app}$$

$$\frac{\Delta \vdash I \overset{\Rightarrow}{\in} R_1 \,\&\, R_2}{\Delta \vdash I \overset{\Rightarrow}{\in} R_1}\,\mathsf{si\_inter1} \qquad \frac{\Delta \vdash I \overset{\Rightarrow}{\in} R_1 \,\&\, R_2}{\Delta \vdash I \overset{\Rightarrow}{\in} R_2}\,\mathsf{si\_inter2} \qquad \frac{R \text{ in } L \qquad \Delta \vdash C \overset{\Leftarrow}{\in} R}{\Delta \vdash (C{\in}L) \overset{\Rightarrow}{\in} R}\,\mathsf{si\_annot}$$

. . . . . .

$$\frac{\Delta, x{\in}R \vdash C \overset{\Leftarrow}{\in} S}{\Delta \vdash \lambda x.C \overset{\Leftarrow}{\in} R \to S}\,\mathsf{sc\_lam} \qquad \frac{\Delta \vdash I \overset{\Rightarrow}{\in} R \qquad |R| \trianglelefteq |S|}{\Delta \vdash I \overset{\Leftarrow}{\in} S}\,\mathsf{sc\_subs}$$

$$\frac{\Delta \vdash \lambda x{:}A.C \overset{\Leftarrow}{\in} R \qquad \Delta \vdash \lambda x{:}A.C \overset{\Leftarrow}{\in} S}{\Delta \vdash \lambda x{:}A.C \overset{\Leftarrow}{\in} R \,\&\, S}\,\mathsf{sc\_inter} \qquad \frac{}{\Delta \vdash \lambda x{:}A_1.C \overset{\Leftarrow}{\in} \top^{A_1 \to A_2}}\,\mathsf{sc\_top}$$

These rules are slightly more non-deterministic than necessary: in the implementation the rules si_inter1 and si_inter2 are never used as the last step in the first premise of the rule sc_subs. This is a minor optimization, and clearly does not affect the correctness of the algorithm. Also, the implementation actually only works with simplified sorts, so the simplification in the rule sc_subs is not needed. We have presented the algorithm without the restriction to unsimplified sorts because it does not seem to add any additional complexity (unlike algorithmic subsorting in Section 3.7).

3.11.3 Soundness and completeness

We cannot directly check the soundness and completeness of this algorithm with respect to the declarative sorting rules: there is no declarative rule for an annotated term ($C{\in}L$). Two possible solutions to this are:

1. Relate the declarative and algorithmic systems via an erasure function which removes all annotations from a checkable term, and an annotation function which produces a checkable term from a declarative sorting derivation.

2. Extend the declarative system with a rule for annotations, and show that the algorithmic and extended declarative systems are equivalent for checkable terms.

We choose the second solution (as in the previous chapter) because the first would leave us without a declarative system for annotated terms. This would be somewhat unsatisfactory: when considering whether an annotated term is correct the programmer would either need to follow the sort checking algorithm exactly, or make sure that the term is annotated exactly as specified by the annotation function. The declarative system allows more flexible forms of reasoning about sort correctness, and since the programmer needs to reason about the correctness of annotated terms, at least in an informal way, it is useful to extend the declarative system to include annotations.

This allows the programmer some flexibility in exactly where they add annotations: they need only ensure that their annotated program is declaratively sort correct, and that it has enough annotations to satisfy the grammar for a checkable term. We will still be interested in an annotation function though: in this context it guarantees that there is always some way of adding annotations to a declaratively well-sorted term.

We thus extend the language of declarative terms, typing rules and the declarative sorting rules as follows.

$$\text{Terms} \quad M ::= \quad \ldots \mid (M{\in}L)$$

$$\frac{R_1 \sqsubset A \quad \ldots \quad R_n \sqsubset A \qquad \Gamma \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma (M{\in}R_1, \ldots, R_n) : A}\ \mathsf{tp\_annot} \qquad \frac{R \text{ in } L \qquad \Delta \vdash M \in R}{\Delta \vdash (M{\in}L) \in R}\ \mathsf{sa\_annot}$$

Annotations are only used during sort-checking, so there is no reason to extend the whole development in this chapter to terms with annotations. Instead we define the function $\|\cdot\|$ that removes annotations from a term, reflecting our intention that sort annotations should be removed before considering terms in other contexts.

$$\begin{array}{rcl} \|M{\in}L\| & = & \|M\| \\ \|x\| & = & x \\ \|c\| & = & c \\ \|\lambda x{:}A.M\| & = & \lambda x{:}A.\|M\| \\ \|M\,N\| & = & \|M\|\,\|N\| \end{array}$$

We now demonstrate that the function $\|\cdot\|$ preserves types and sorts, justifying the notion that annotations can be removed once sort checking has been done. We will need the following lemma.

Lemma 3.11.1 (Value Erasure)
If M is a value then $\|M\|$ is a value.

Proof: By a straightforward induction on the structure of M. Only the case for $V^{\mathsf{a}}\, V$ requires the induction. □

Lemma 3.11.2 (Typing Erasure)
If $\Gamma \vdash M : A$ then $\Gamma \vdash \|M\| : A$.

Proof: By a straightforward induction on the structure of the derivation. We show the case for the rule tp_annot. The remaining cases simply rebuild the derivation by mirroring the structure of the given derivation.

Case:
$$\frac{R_1 \sqsubset A \quad \ldots \quad R_n \sqsubset A \qquad \overset{\mathcal{E}}{\Gamma \vdash N : A}}{\Gamma \vdash (N{\in}R_1, \ldots, R_n) : A}\ \mathsf{tp\_annot}.$$

Then $\|M\| = \|N{\in}L\| = \|N\|$ and we apply the induction hypothesis to $\mathcal{E}$ to obtain $\Gamma \vdash \|N\| : A$, as required.

□

Lemma 3.11.3 (Sorting Erasure)
If $\Delta \vdash M \in R$ then $\Delta \vdash \|M\| \in R$.

Proof: By a straightforward induction on the structure of the derivation. We show the cases for rules sa_annot and sa_inter. The remaining cases simply rebuild the derivation by mirroring the structure of the given derivation.

Case:
$$\frac{R \text{ in } L \qquad \overset{\mathcal{D}}{\Delta \vdash N \in R}}{\Delta \vdash (N{\in}L) \in R}\ \mathsf{sa\_annot}.$$

Then $\|M\| = \|N{\in}L\| = \|N\|$ and we apply the induction hypothesis to $\mathcal{D}$ to obtain $\Delta \vdash \|N\| \in R$, as required.

Case:
$$\frac{\overset{\mathcal{D}_1}{\Delta \vdash V \in R_1} \qquad \overset{\mathcal{D}_2}{\Delta \vdash V \in R_2}}{\Delta \vdash V \in R_1 \,\&\, R_2}\ \mathsf{sa\_inter}.$$

Then $\|M\| = \|V\|$. We apply the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ to obtain $\Delta \vdash \|V\| \in R_1$ and $\Delta \vdash \|V\| \in R_1$. We then apply rule sa_inter to rebuild the derivation, using the above lemma to satisfy the requirement that $\|V\|$ is a value.

□

We now show that for inferable and checkable terms our algorithm is correct with respect to this extended declarative system. In Section 3.12 we will extend this result to correctness with respect to the original declarative system by relating annotated and unannotated terms in the declarative system via an annotation function.

The proof of the soundness result is relatively straightforward: each of the algorithmic sort checking rules corresponds to a derivable rule for the declarative sorting judgment. Thus algorithmic sort derivations correspond to a fragment of the declarative sort derivations.

Theorem 3.11.4 (Soundness of Sort Checking)

1. *If* $\Delta \vdash I \stackrel{\Rightarrow}{\in} R$ *then* $\Delta \vdash I \in R$.
2. *If* $\Delta \vdash C \stackrel{\Leftarrow}{\in} S$ *then* $\Delta \vdash C \in S$.

Proof:
By simultaneous structural induction on the derivations $\mathcal{D} :: \Delta \vdash I \stackrel{\Rightarrow}{\in} R$ and $\mathcal{E} :: \Delta \vdash C \stackrel{\Leftarrow}{\in} S$.

We have the following cases for part 1.

Case: $\mathcal{D}$ is an instance of si_var or si_const. Immediate using rule sa_var or sa_const.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E}_2 \\ \Delta \vdash I \stackrel{\Rightarrow}{\in} R_2 \to R & \Delta \vdash C \stackrel{\Leftarrow}{\in} R_2\end{array}}{\Delta \vdash I\, C \stackrel{\Rightarrow}{\in} R}\ \mathsf{si\_app}.$

$\Delta \vdash I \in R_2 \to R$ — By ind. hyp. on $\mathcal{D}_1$
$\Delta \vdash C \in R_2$ — By ind. hyp. on $\mathcal{E}_2$
$\Delta \vdash I\, C \in R$ — By rule sa_app

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Delta \vdash I \stackrel{\Rightarrow}{\in} R_1 \,\&\, R_2\end{array}}{\Delta \vdash I \stackrel{\Rightarrow}{\in} R_1}\ \mathsf{si\_inter1}.$

$\Delta \vdash I \in R_1 \,\&\, R_2$ — By ind. hyp. on $\mathcal{D}_1$
$R_1 \,\&\, R_2 \leq R_1$ — By rule sub_inter_left_1
$\Delta \vdash I \in R_1$ — By rule sa_subs

Case: $\mathcal{D}$ is an instance of si_inter2. Symmetric to the previous case.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}R \text{ in } L & \mathcal{E}_1 \\ & \Delta \vdash C \stackrel{\Leftarrow}{\in} R\end{array}}{\Delta \vdash (C \in L) \stackrel{\Rightarrow}{\in} R}\ \mathsf{si\_annot}.$

$\Delta \vdash C \in R$ — By ind. hyp. on $\mathcal{E}_1$
$\Delta \vdash (C \in L) \in R$ — By rule sa_annot

For part 2 we have the following cases.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{D}_1}{\Delta \vdash I \overset{\Rightarrow}{\in} S} \qquad |S| \trianglelefteq |R|}{\Delta \vdash I \overset{\Leftarrow}{\in} R}\,\mathsf{sc\_subs}.$

$\Delta \vdash I \in S$ By ind. hyp. on $\mathcal{D}_1$
$S \leq R$ By Theorem 3.7.5 (Correctness of $\trianglelefteq$)
$\Delta \vdash I \in R$ By rule sa_subs

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Delta \vdash \lambda x{:}A.C \overset{\Leftarrow}{\in} R_1} \qquad \overset{\mathcal{E}_2}{\Delta \vdash \lambda x{:}A.C \overset{\Leftarrow}{\in} R_2}}{\Delta \vdash \lambda x{:}A.M \overset{\Leftarrow}{\in} R_1 \,\&\, R_2}\,\mathsf{sc\_inter}.$

By the induction hypothesis on $\mathcal{E}_1$ and $\mathcal{E}_2$ followed by rule sa_inter.

Case: $\mathcal{E} = \dfrac{}{\Delta \vdash \lambda x{:}A.M \overset{\Leftarrow}{\in} \top^A}\,\mathsf{sc\_top}.$

By rule sa_top

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Delta, x{\in}R_1 \vdash M \overset{\Leftarrow}{\in} R_2}}{\Delta \vdash \lambda x{:}A.M \overset{\Leftarrow}{\in} R_1 \to R_2}\,\mathsf{sc\_lam}.$

By the induction hypothesis on $\mathcal{E}_1$ followed by rule sa_lam.

□

The proof of completeness of sort checking is a little more difficult: the declarative sorting rules allow derivations to be structured in ways that cannot be directly mirrored in the algorithmic system. However, the strong inversion properties demonstrated in Section 3.10 allow a reasonably direct proof by induction on the structure of terms. The following lemma extends these properties to annotated terms.

Lemma 3.11.5 (Inversion for Annotations)
$\Delta \vdash (M{\in}L) \in S$ *if and only if* $R \leq S$ *and* $\Delta \vdash M \in R$ *for some* R *in* L.

Proof:
The "if" part is a simple consequence of rules sa_annot and sa_subs. The "only if" part is proved by structural induction on the derivation $\mathcal{D} :: \Delta \vdash (M{\in}L) \in S$. Annotated terms are not considered to be values, so we have only the following two cases.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{E}_1}{S \text{ in } L} \qquad \overset{\mathcal{D}_2}{\Delta \vdash M \in S}}{\Delta \vdash (M{\in}L) \in S}\,\mathsf{sa\_annot}.$

We have the required result with $R = S$, since $S \leq S$ (by sub_reflex) and $\mathcal{E}_1$, $\mathcal{D}_2$ satisfy the remaining two requirements.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E}_2 \\ \Delta \vdash (M{\in}L) \in S' & S' \leq S\end{array}}{\Delta \vdash (M{\in}L) \in S}\ \mathsf{sa\_sub}.$

$\Delta \vdash (M{\in}L) \in R'$	and	
$R' \leq S'$	for some R' in L	By ind. hyp. on $\mathcal{D}_1$
$R' \leq S$		By rule sub_trans using $\mathcal{E}_2$

As required, with $R = R'$.

□

We will also require the following lemma in the case for an application in the completeness proof.

Lemma 3.11.6 (Completeness of $\stackrel{\Rightarrow}{\in}$ for $\rightarrow$)

If $\Delta \vdash I \stackrel{\Rightarrow}{\in} R$ *and* $|R| \trianglelefteq |S_1| \rightarrow |S_2|$ *then*

for some R_1, R_2, *we have* $\Delta \vdash I \stackrel{\Rightarrow}{\in} R_1 \rightarrow R_2$ *and* $S_1 \leq R_1$ *and* $R_2 \leq S_2$.

Proof: By induction on the structure of $\mathcal{D} :: |R| \trianglelefteq |S_1| \rightarrow |S_2|$. The case for subalg_arrow is immediate, via the correctness of algorithmic subsorting. The cases for subalg_&L1 and subalg_&L2 simply refer to the induction hypothesis. □

We use the following slightly generalized form for the completeness theorem to support the induction in the proof.

Theorem 3.11.7 (Completeness of Sort Checking)

1. *If* $\Delta \vdash I \in R$ *then there is some* R' *such that* $\Delta \vdash I \stackrel{\Rightarrow}{\in} R'$ *and* $R' \leq R$.
2. *If* $\Delta \vdash C \in R$ *and* $R \leq R'$ *then* $\Delta \vdash C \stackrel{\Leftarrow}{\in} R'$.

Proof:
By structural induction on I and C. We order Part 1 of the theorem before Part 2, i.e. in the proof of Part 2 when C is an inferable term we allow an appeal to Part 1 of the induction hypothesis for the same term, but not vice-versa.

We have the following cases for I in Part 1 of the theorem.

Case: $I = I_1\, C_2$.

$\Delta \vdash I_1\, C_2 \in R$		Assumption
$\Delta \vdash I_1 \in R_2 \rightarrow R$	and	
$\Delta \vdash C_2 \in R_2$	for some R_2	By Inv. Lem. (3.10.2)
$\Delta \vdash I_1 \stackrel{\Rightarrow}{\in} R_1'$	for some $R_1' \leq R_2 \rightarrow R$	By ind. hyp. (1)
$\Delta \vdash I_1 \stackrel{\Rightarrow}{\in} S_2 \rightarrow S$	and	
$R_2 \leq S_2$ and $S \leq R$	for some S_2, S	By Compl.-$\stackrel{\Rightarrow}{\in}$-$\rightarrow$ (3.11.6)
$\Delta \vdash C_2 \stackrel{\Leftarrow}{\in} S_2$		By ind. hyp. (2)
$\Delta \vdash I_1\, C_2 \stackrel{\Rightarrow}{\in} S$		By rule si_app

As required, with $R' = S$.

Case: $I = x$.

$\Delta \vdash x \in R$	Assumption
$x{\in}S$ in Δ for some $S \leq R$	By Inv. Lemma (3.10.3)
$\Delta \vdash x \overset{\Rightarrow}{\in} S$	By rule si_var

As required, with $R' = S$.

Case: $I = c$.

Similar to the previous case.

Case: $I = (C{\in}L)$.

$\Delta \vdash (C{\in}L) \in R$	Assumption
$\Delta \vdash C \in S$ and	
$S \leq R$ for some S in L	By Inv. Lemma (3.11.5)
$\Delta \vdash C \overset{\Leftarrow}{\in} S$	By ind. hyp. (2)
$\Delta \vdash (C{\in}L) \overset{\Rightarrow}{\in} S$	By rule si_annot

As required, with $R' = S$.

Case: $I = \lambda x{:}A_1.C$. Cannot occur, because Part 1 requires an inferable term.

We have the following two cases for C in Part 2 of the theorem.

Case: C is an inferable term.

$\Delta \vdash C \overset{\Rightarrow}{\in} S'$ for some $S' \leq R$	By ind. hyp. (1)
$S' \leq R'$	By rule sub_trans
$\Delta \vdash C \overset{\Leftarrow}{\in} R'$	By rule sc_subs

Case: $C = \lambda x{:}A_1.C_2$.

Then

$\Delta \vdash \lambda x{:}A_1.C_2 \in R$	Assumption
$R \leq R'$	Assumption

We now need to prove that $\Delta \vdash \lambda x{:}A_1.C_2 \overset{\Leftarrow}{\in} R'$. We do this by proving the more general result that for any S' such that $R \leq S'$ we have $\Delta \vdash \lambda x{:}A_1.C_2 \overset{\Leftarrow}{\in} S'$ (taking the instance $S' = R'$ gives the required result). We use a nested induction on S'.

We have three sub-cases for S' in this nested induction.

Subcase: $S' = S'_1 \to S'_2$.

$R \leq S'_1 \to S'_2$	Assumption
$\Delta, x{\in}S'_1 \vdash C_2 \in S'_2$	By Inv. Lemma for λ (3.8.5)
$\Delta, x{\in}S'_1 \vdash C_2 \overset{\Leftarrow}{\in} S'_2$	By ind. hyp. (2)
$\Delta \vdash \lambda x{:}A_1.C_2 \overset{\Leftarrow}{\in} S'_1 \to S'_2$	By rule sc_lam

Subcase: $S' = S'_1 \,\&\, S'_2$.

$R \leq S'_1 \,\&\, S'_2$	Assumption
$R \leq S'_1$ and $R \leq S'_2$	By rules sub_trans, sub_left_1,2
$\Delta \vdash \lambda x{:}A_1.C_2 \overset{\Leftarrow}{\in} S'_1$	By nested ind. hyp.
$\Delta \vdash \lambda x{:}A_1.C_2 \overset{\Leftarrow}{\in} S'_2$	By nested ind. hyp.
$\Delta \vdash \lambda x{:}A_1.C_2 \overset{\Leftarrow}{\in} S'_1 \,\&\, S'_2$	By rule sc_inter

Subcase: $S' = \top^{A_1 \to A_2}$.

Immediate using rule sc_top.

□

The use of the inversion lemma for λ-abstractions (Lemma 3.8.5) here is carefully designed to avoid the need to generalize the induction hypothesis further to sort contexts containing subsorts, i.e. $\Delta \leq \Delta'$. Such a generalization seems possible, but results in a more involved proof. Also, our use of inversion lemmas does not allow us to perform a proof by induction on the structure of derivations, unlike the use of ordinary inversion: the derivations obtained by the lemmas may not be sub-derivations. Thus, the induction is on the structure of terms, and to support this the inversion lemmas are designed to produce appropriate information about subterms.

3.12 Annotatability

We now demonstrate that every well-sorted unannotated term can be annotated to produce a checkable term. We follow the technique used in Section 2.11 closely, although we present a few more details here.

We have the same definitions of minimal inferable terms and minimal checkable terms as in Section 2.11, which are as follows.

Definition 3.12.1

$$\begin{array}{rrcl} \textit{Minimal Inferable Terms} & \overset{\mathsf{m}}{I} & ::= & c \mid x \mid \overset{\mathsf{m}}{I}\,\overset{\mathsf{m}}{C} \mid ((\lambda x{:}A.\overset{\mathsf{m}}{C}) \in L) \\ \textit{Minimal Checkable Terms} & \overset{\mathsf{m}}{C} & ::= & c \mid x \mid \overset{\mathsf{m}}{I}\,\overset{\mathsf{m}}{C} \mid \lambda x{:}A.\overset{\mathsf{m}}{C} \end{array}$$

An annotated term $((\lambda x{:}A.\overset{\mathsf{m}}{C}) \in L)$ is not a value, so when a function is assigned an intersection sort like $(R_1 \to R_2) \,\&\, (S_1 \to S_2)$ the intersection must appear as one of the alternatives in the annotation L: it is not sufficient for each conjunct of the intersection to be present. This is unlike the situation in Chapter 2, but we did not make any use of this fact when constructing annotated terms in Section 2.11, and in fact we construct annotations in essentially the same way here.

We have the same functions for combining annotations as in the previous chapter, which are as follows.

$$
\begin{array}{rcl}
c \overset{\mathsf{i}}{\bowtie} c & = & c \\
x \overset{\mathsf{i}}{\bowtie} x & = & x \\
(\overset{\mathsf{m}}{I_1}\ \overset{\mathsf{m}}{C_1}) \overset{\mathsf{i}}{\bowtie} (\overset{\mathsf{m}}{I_2}\ \overset{\mathsf{m}}{C_1}) & = & (\overset{\mathsf{m}}{I_1} \overset{\mathsf{i}}{\bowtie} \overset{\mathsf{m}}{I_2})\ (\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2}) \\
((\lambda x{:}A.\overset{\mathsf{m}}{C_1}) \in L_1) \overset{\mathsf{i}}{\bowtie} ((\lambda x{:}A.\overset{\mathsf{m}}{C_2}) \in L_2) & = & ((\lambda x{:}A.\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2}) \in L_1, L_2)
\end{array}
$$

$$
\begin{array}{rcl}
c \overset{\mathsf{c}}{\bowtie} c & = & c \\
x \overset{\mathsf{c}}{\bowtie} x & = & x \\
(\overset{\mathsf{m}}{I_1}\ \overset{\mathsf{m}}{C_1}) \overset{\mathsf{c}}{\bowtie} (\overset{\mathsf{m}}{I_2}\ \overset{\mathsf{m}}{C_2}) & = & (\overset{\mathsf{m}}{I_1} \overset{\mathsf{i}}{\bowtie} \overset{\mathsf{m}}{I_2})\ (\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2}) \\
(\lambda x{:}A.\overset{\mathsf{m}}{C_1}) \overset{\mathsf{c}}{\bowtie} (\lambda x{:}A.\overset{\mathsf{m}}{C_2}) & = & (\lambda x{:}A.\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2})
\end{array}
$$

The following lemma demonstrates that these functions have the intended properties, and is exactly like that in the previous chapter, except that the sort assignment system is different in this chapter.

Lemma 3.12.2 (Annotation Combination)

1. *If* $\|\overset{\mathsf{m}}{I_1}\| = \|\overset{\mathsf{m}}{I_2}\|$ *and either* $\Delta \vdash \overset{\mathsf{m}}{I_1} \in R$ *or* $\Delta \vdash \overset{\mathsf{m}}{I_2} \in R$ *then* $\Delta \vdash (\overset{\mathsf{m}}{I_1} \overset{\mathsf{i}}{\bowtie} \overset{\mathsf{m}}{I_2}) \in R$.
2. *If* $\|\overset{\mathsf{m}}{C_1}\| = \|\overset{\mathsf{m}}{C_2}\|$ *and either* $\Delta \vdash \overset{\mathsf{m}}{C_1} \in R$ *or* $\Delta \vdash \overset{\mathsf{m}}{C_2} \in R$ *then* $\Delta \vdash (\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2}) \in R$.

Proof: By structural induction on the sorting derivations $\mathcal{D}_1 :: \Delta \vdash \overset{\mathsf{m}}{I_1} \in R$, $\mathcal{D}_2 :: \Delta \vdash \overset{\mathsf{m}}{I_2} \in R$, $\mathcal{E}_1 :: \Delta \vdash \overset{\mathsf{m}}{C_1} \in R$, and $\mathcal{E}_2 :: \Delta \vdash \overset{\mathsf{m}}{C_2} \in R$.

We focus on the cases for $\mathcal{D}_1$ and $\mathcal{E}_1$ since the other two are symmetric. We show two cases: the remaining cases are similar and straightforward.

Case: $\mathcal{D}_1 = \dfrac{R \text{ in } L_1 \qquad \Delta \vdash \overset{\mathsf{m}}{C_{11}} \in R \;(\mathcal{D}_{11})}{\Delta \vdash (\overset{\mathsf{m}}{C_{11}} \in L_1) \in R}\ \mathsf{sa\_annot}.$

$(\overset{\mathsf{m}}{C_{11}} \in L_1) \overset{\mathsf{i}}{\bowtie} (\overset{\mathsf{m}}{C_{22}} \in L_2) = ((\overset{\mathsf{m}}{C_{11}} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_{22}}) \in L_1, L_2)$	By def. $\overset{\mathsf{i}}{\bowtie}, \overset{\mathsf{c}}{\bowtie}$
$\Delta \vdash (\overset{\mathsf{m}}{C_{11}} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_{22}}) \in R$	By ind. hyp. on $\mathcal{D}_{11}$
R in L_1, L_2	By def. L_1, L_2
$\Delta \vdash ((\overset{\mathsf{m}}{C_{11}} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_{22}}) \in L_1, L_2) \in R$	By rule $\mathsf{sa\_annot}$

Case: $\mathcal{E}_1 = \dfrac{\begin{array}{cc}\mathcal{E}_{11} & \mathcal{E}_{12} \\ \Delta \vdash \overset{\mathsf{m}}{C}_1 \in R_1 & \Delta \vdash \overset{\mathsf{m}}{C}_1 \in R_2\end{array}}{\Delta \vdash \overset{\mathsf{m}}{C}_1 \in R_1 \,\&\, R_2}\ \mathsf{sa\_inter}$

with $\overset{\mathsf{m}}{C}_1$ a value.

$\overset{\mathsf{m}}{C}_1 = V^{\mathsf{a}}$ or x or $\lambda x{:}A.\overset{\mathsf{m}}{C}_{11}$	By def. $\overset{\mathsf{m}}{C}$, values
$\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2$ is a value	By def. $\overset{\mathsf{c}}{\bowtie}$, values
$\Delta \vdash (\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2) \in R_1$	By ind. hyp. on $\mathcal{E}_1$
$\Delta \vdash (\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2) \in R_2$	By ind. hyp. on $\mathcal{E}_2$
$\Delta \vdash (\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2) \in R_1 \,\&\, R_2$	By rule $\mathsf{sa\_inter}$

□

Theorem 3.12.3 (Annotatability)

If $\Delta \vdash M \in R$ then we can construct a minimal inferable term $\overset{\mathsf{m}}{I}$ and a minimal checkable term $\overset{\mathsf{m}}{C}$ such that $\|\overset{\mathsf{m}}{I}\| = M$ and $\Delta \vdash \overset{\mathsf{m}}{I} \in R$ and $\|\overset{\mathsf{m}}{C}\| = M$ and $\Delta \vdash \overset{\mathsf{m}}{C} \in R$.

Proof: By induction on the sorting derivation. We show two cases. The remaining cases simply rebuild the term, using the induction hypothesis on sub-derivations.

Case: $\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash V \in R_1 & \Delta \vdash V \in R_2\end{array}}{\Delta \vdash V \in R_1 \,\&\, R_2}\ \mathsf{sa\_inter}.$

If $V = c$ or $V = x$ then V already has the required forms, so we set $\overset{\mathsf{m}}{I} = V$ and $\overset{\mathsf{m}}{C} = V$.

If $V = V^{\mathsf{a}}\, V$ then we apply the induction hypothesis to V^{a} and V and then rebuild the required terms from the results.

Otherwise $V = \lambda x{:}A.M_1$, and then:

$\Delta \vdash \overset{\mathsf{m}}{C}_1 \in R_1$	and	
$\|\overset{\mathsf{m}}{C}_1\| = V$	for some $\overset{\mathsf{m}}{C}_1$	By ind. hyp. on $\mathcal{D}_1$
$\Delta \vdash \overset{\mathsf{m}}{C}_2 \in R_2$	and	
$\|\overset{\mathsf{m}}{C}_2\| = V$	for some $\overset{\mathsf{m}}{C}_2$	By ind. hyp. on $\mathcal{D}_2$
$\Delta \vdash \overset{\mathsf{m}}{C} \in R_1$	and	
$\Delta \vdash \overset{\mathsf{m}}{C} \in R_2$	and	
$\|\overset{\mathsf{m}}{C}\| = V$	for $\overset{\mathsf{m}}{C} = \overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2$	By above lemma
$\overset{\mathsf{m}}{C} = \lambda x{:}A.\overset{\mathsf{m}}{C}_3$		By def. $\|\cdot\|$
$\overset{\mathsf{m}}{C}$ is a value		By def. value
$\Delta \vdash \overset{\mathsf{m}}{C} \in R_1 \,\&\, R_2$		By rule $\mathsf{sa\_inter}$
$\Delta \vdash (\overset{\mathsf{m}}{C} \in R_1 \,\&\, R_2) \in R_1 \,\&\, R_2$		By rule $\mathsf{sa\_annot}$
$\|(\overset{\mathsf{m}}{C} \in R_1 \,\&\, R_2)\| = \|\overset{\mathsf{m}}{C}\| = M$		By def. $\|\cdot\|$

Then $\overset{\mathsf{m}}{C}$ is as required, and we set $\overset{\mathsf{m}}{I} = (\overset{\mathsf{m}}{C} \in R_1 \,\&\, R_2)$.

Case: $$\dfrac{\begin{array}{c}\mathcal{D}_2\\ \Delta, x{\in}R_1 \vdash M_2 \in R_2\end{array}}{\Delta \vdash \lambda x{:}A.M_2 \in R_1 \to R_2}\ \mathsf{sa\_lam}.$$

$\Delta, x{\in}R_1 \vdash \overset{\mathsf{m}}{C}_2 \in R_2$	and	
$\|\overset{\mathsf{m}}{C}_2\| = M_2$	for some $\overset{\mathsf{m}}{C}_2$	By ind. hyp. on $\mathcal{D}_2$
$\Delta \vdash \lambda x{:}A.\overset{\mathsf{m}}{C}_2 \in R_1 \to R_2$		By rule sa_lam
$\Delta \vdash ((\lambda x{:}A.\overset{\mathsf{m}}{C}_2) \in R_1 \to R_2) \in R_1 \to R_2$		By rule sa_annot

Then $\|(\lambda x{:}A.\overset{\mathsf{m}}{C}_2) \in R_1 \to R_2)\| = \|\lambda x{:}A.\overset{\mathsf{m}}{C}_2\| = \lambda x{:}A.\|\overset{\mathsf{m}}{C}_2\| = \lambda x{:}A.M_2$
and we choose $\overset{\mathsf{m}}{C} = \lambda x{:}A.\overset{\mathsf{m}}{C}_2$ and $\overset{\mathsf{m}}{I} = ((\lambda x{:}A.\overset{\mathsf{m}}{C}_2) \in R_1 \to R_2)$.

□

We conclude by observing that if we can assign a sort to a term, then we can construct appropriate annotations of the term which allow that sort to be verified by the sort checking algorithm. This result combines two main results of this chapter, and is expressed formally as follows.

Corollary 3.12.4 *If $\Delta \vdash M \in R$ then we can construct a checkable term C and an inferable term I such that $\Delta \vdash C \overset{\Leftarrow}{\in} R$ and there is some $R' \leq R$ such that $\Delta \vdash I \overset{\Rightarrow}{\in} R'$.*

Proof: By composing the previous theorem with the Completeness of Sort Checking Theorem (3.11.7). □

Chapter 4

Soundness with effects

In this chapter we demonstrate that the restrictions presented in Chapter 3 lead to a system that is sound in the presence of effects. We do this by considering a small call-by-value language $ML^{\&\mathbf{ref}}$ with a typical feature involving effects, namely mutable reference cells. Following Chapter 3, we place a value restriction on the introduction of intersections, and omit the problematic distributivity subtyping rules. We then show that this leads to a sound system by proving a progress and type preservation theorem for $ML^{\&\mathbf{ref}}$. An analysis of the proof gives some insight as to why each of our restrictions is required.

This theorem ensures that the unsoundness demonstrated in the examples at the start of Chapter 3 cannot occur in $ML^{\&\mathbf{ref}}$. We repeat these examples here. The first uses intersection introduction for a non-value.

```
(*[ cell <: (nat ref) & (pos ref) ]*)
val cell = ref one
val () = (cell := zero)

(*[ result <: pos ]*)
val result = !cell
```

The second example uses the distributivity rule for intersections.

```
(*[ f <: (unit -> (pos ref)) & (unit -> (nat ref)) ]*)
fun f () = ref one

(*[ cell <: (pos ref) & (nat ref) ]*)
val cell = f ()
```

Proving a progress result would not be convincing unless our language includes sufficient features to express unsound examples like these that may arise in a fully featured language. For example, proving the result for a λ-calculus with references and no other constructs would be somewhat unsatisfying. On the other hand, proving this result for the whole of a fully featured language is likely to be tedious, and the essential ideas of the proof are likely to get lost in the details. Here we have chosen to prove the result for a language which includes an example datatype for bit strings, with subtypes for natural numbers and positive numbers. These correspond to the example datasort declarations in Section 1.4.3, which we repeat here.

```
datatype bits = bnil | b0 of bits | b1 of bits

(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat ]*)
```

Using such an example type may be a little unsatisfying, but it is clear that our proof does not depend on any special properties of it, and so should easily extend to other datatypes. This choice also allows us to consider a specific instance of a language with subtypes of a datatype before considering a general framework for a language parameterized by datatype and datasort declarations in Chapter 5 and Chapter 6. We could have delayed a proof of soundness in the presence of effects until after introducing this general framework, but such a presentation would have left a very large gap between the introduction of the value restriction and the validation of the claim that it results in soundness. It also would have a resulted in the essential ideas of the proof being somewhat harder to follow, due to the generality of the framework.

Unlike previous chapters, we do not include a refinement restriction in this chapter. This means that we include general intersection types in the type system of $ML^{\&\mathbf{ref}}$ instead of restricting them to a level of sorts refining the types. We do this because the refinement restriction is orthogonal to soundness in the presence of effects. Thus, our soundness result is more general than required for refinement types, and would also apply to e.g. operator overloading via intersection types. Treating general intersection types also allows a shorter and simpler presentation, since we do not need separate type and sort levels.

One view of the situation in this chapter is that we have a trivial level of types that assigns the single type T to every term, and a level of sorts that includes all the types of this chapter, with every sort refining T. However, this view does not accurately reflect one important aspect: we consider that the type system of this chapter is necessary to judge the validity of terms prior to considering their semantics, since the semantics of terms only satisfies progress for terms that can be assigned sorts. Thus, by the philosophical distinction between types and sorts outlined in Section 1.3, the types in this chapter are properly types, and not sorts.

Interestingly, our progress theorem holds even for terms which are assigned the type $\top$: this is consequence of restricting the introduction rule for $\top$ to values. In contrast, progress would clearly fail for such terms in a language with the standard rules for $\top$, even without effects: the standard introduction rule allows every term to be assigned $\top$.

The presentation in this chapter closely follows a paper co-authored with Frank Pfenning [DP00].

4.1 Syntax

The syntax of $ML^{\&\mathbf{ref}}$ is relatively standard for a call-by-value language in the ML family. We include fixed-points with eager unrolling, and distinguish two kinds of variables: those bound in λ, **let** and **case** expressions which stand for values (denoted by x), and those bound in **fix** expressions which stand for arbitrary terms (denoted by u). We use identifiers l to address cells in the store during evaluation. We include the term construct **let** $x = M$ **in** N here even though it is equivalent to $(\lambda x.N)\ M$: the latter would not fit well with our approach to bidirectional sort checking (although we do not include sort checking in this chapter).

We also include our example datatype bits for strings of bits, along with two subtypes nat for natural numbers (bit-strings without leading zeroes) and pos for positive natural numbers. We represent natural numbers as bit-strings in standard form, with the least significant bit rightmost and no leading zeroes. We view $\mathsf{0}$ and $\mathsf{1}$ as constructors written in postfix form, and ϵ stands for the empty string. For example, 6 would be represented as $\epsilon\mathsf{110}$. We include an ML-style **case** expression to deconstruct strings of bits.

$$\begin{array}{rrcl}
\text{Types} & A, B & ::= & A \to B \mid A\ \mathbf{ref} \mid \mathsf{1} \mid A \,\&\, B \mid \top \\
& & \mid & \mathsf{bits} \mid \mathsf{nat} \mid \mathsf{pos} \\
\text{Terms} & M, N & ::= & x \mid \lambda x.M \mid M\ N \\
& & \mid & \mathbf{let}\ x = M\ \mathbf{in}\ N \\
& & \mid & u \mid \mathbf{fix}\ u.M \\
& & \mid & l \mid \mathbf{ref}\ M \mid \mathbf{!}\ M \mid M := N \mid () \\
& & \mid & \epsilon \mid M\ \mathsf{0} \mid M\ \mathsf{1} \\
& & \mid & \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ \mathsf{0} \Rightarrow M_2 \mid y\ \mathsf{1} \Rightarrow M_3
\end{array}$$

As in previous chapters, we write $\{N/x\}M$ for the result of substituting N for x in M. In this chapter it is important that we use this notation rather than the more standard $[N/x]M$, in order to avoid confusion with the standard notation for evaluation contexts $E[M]$ which we will use in the reduction semantics.

We distinguish the following terms as *values*. We do not include expression variables u because during evaluation these may be replaced by non-values. Unlike Section 3.3 we do not need a definition of atomic values: applications of the constructors $\mathsf{1}$ and $\mathsf{0}$ are built directly into the language.

$$\text{Values} \quad V \quad ::= \quad x \mid \lambda x.M \mid l \mid () \mid \epsilon \mid V\ \mathsf{0} \mid V\ \mathsf{1}$$

For the typing judgment, we need to assign types to variables and cells in contexts Γ and Ψ, respectively. Moreover, during execution of a program we need to maintain a store C.

$$\begin{array}{rrcl}
\text{Variable Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, u{:}A \\
\text{Cell Contexts} & \Psi & ::= & \cdot \mid \Psi, l{:}A \\
\text{Store} & C & ::= & \cdot \mid C, (l = V) \\
\text{Program States} & P & ::= & C \triangleright M
\end{array}$$

We assume that variables x, u and cells l can be declared at most once in a context or store. We omit leading $\cdot$'s from contexts, and write Γ, Γ' for the result of appending two variable disjoint contexts (and similarly for cell contexts and stores).

The following is an example of a program using the syntax of $ML^{\&\mathbf{ref}}$. It corresponds to the first counterexample from Chapter 3 (which was repeated at the start of this chapter), except that it contains no type annotations.

$$\begin{array}{l}
\mathbf{let}\ x_{\mathsf{cell}} = \mathbf{ref}(\epsilon\,\mathsf{1}) \\
\mathbf{in\ let}\ y = (x_{\mathsf{cell}} := \epsilon) \\
\mathbf{in\ let}\ x_{\mathsf{result}} = \mathbf{!}\ x_{\mathsf{cell}} \\
\mathbf{in}\ x_{\mathsf{result}}
\end{array}$$

The type assignment system presented later in this chapter allows us to assign this program the type nat but correctly prevents us from assigning the type pos.

4.2 Subtyping

The subtyping judgment for this language has the form.

$A \leq B$ Type A is a subtype of B.

Following Chapter 3, we have the standard rules for intersection types with the exception of the distributivity subtyping rules. We also have inclusions between the base types bits, nat and pos, which we build directly into the the subtyping judgment. The **ref** type constructor is non-variant.

$$\frac{}{A \leq A} \qquad \frac{A_1 \leq A_2 \quad A_2 \leq A_3}{A_1 \leq A_3}$$

$$\frac{}{A_1 \,\&\, A_2 \leq A_1} \qquad \frac{}{A_1 \,\&\, A_2 \leq A_2} \qquad \frac{A \leq B_1 \quad A \leq B_2}{A \leq B_1 \,\&\, B_2} \qquad \frac{}{A \leq \top}$$

$$\frac{}{\mathsf{pos} \leq \mathsf{nat}} \qquad \frac{}{\mathsf{nat} \leq \mathsf{bits}} \qquad \frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2} \qquad \frac{A \leq B \quad B \leq A}{A \;\mathbf{ref} \leq B \;\mathbf{ref}}$$

We obtain an algorithmic version of subtyping roughly by following the subsorting algorithm in Chapter 3. We differ by not restricting the algorithm to simplified sorts, which is unnecessary for the current lattice of base types.

We use the notation A^{n} for an *non-intersection type*, namely one that does not have an intersection as the top constructor, although it may contain embedded intersections (similar to the notation R^{n} used in Section 2.10).

$A \trianglelefteq B$ Type A is determined to be a subtype of type B.

$$\frac{}{\mathsf{pos} \trianglelefteq \mathsf{pos}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{bits}}$$

$$\frac{}{\mathsf{nat} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{nat} \trianglelefteq \mathsf{bits}} \qquad \frac{}{\mathsf{bits} \trianglelefteq \mathsf{bits}}$$

$$\frac{B_1 \trianglelefteq A_1 \qquad A_2 \trianglelefteq B_2}{A_1 \to A_2 \trianglelefteq B_1 \to B_2} \qquad \frac{A \trianglelefteq B \qquad B \trianglelefteq A}{A \;\mathbf{ref} \trianglelefteq B \;\mathbf{ref}} \qquad \frac{}{1 \trianglelefteq 1}$$

$$\frac{A_1 \trianglelefteq B^{\mathsf{n}}}{A_1 \,\&\, A_2 \trianglelefteq B^{\mathsf{n}}} \qquad \frac{A_2 \trianglelefteq B^{\mathsf{n}}}{A_1 \,\&\, A_2 \trianglelefteq B^{\mathsf{n}}} \qquad \frac{A \trianglelefteq B_1 \qquad A \trianglelefteq B_1}{A \trianglelefteq B_1 \,\&\, B_2} \qquad \frac{}{A \trianglelefteq \top}$$

We now prove three properties and show that algorithmic and declarative subtyping coincide.

The properties and proofs are essentially the same as in Chapter 3, and in particular the lack of the refinement restriction does not result in any new complications.

Lemma 4.2.1 (Properties of Algorithmic Subtyping)
The algorithmic subtyping judgment satisfies:

1. *If* $A \trianglelefteq B$ *then* $A \,\&\, A' \trianglelefteq B$ *and* $A' \,\&\, A \trianglelefteq B$.
2. $A \trianglelefteq A$.
3. *If* $A_1 \trianglelefteq A_2$ *and* $A_2 \trianglelefteq A_3$ *then* $A_1 \trianglelefteq A_3$.

Proof: By simple inductions on the given types and derivations. □

Theorem 4.2.2 $A \trianglelefteq B$ *if and only if* $A \leq B$.

Proof: In each direction, by induction on the given derivation, using the properties in the preceding lemma. □

4.3 Typing of terms

The typing judgment for terms has the form:

$\Psi; \Gamma \vdash M : A$ Term M has type A in cell context Ψ and variable context Γ.

The typing rules are given in Figure 4.1.

These rules are standard for functions, let definitions, fixed points, references, and intersection types, with the exception that the introduction rules for & and $\top$ are restricted to values. There are three typing rules for **case**, depending on whether the subject is assigned type bits, nat, or pos. Note that the branch for ϵ does not need to be checked when the case subject has type pos.

The structural properties of weakening, exchange and contraction from Chapter 3 extend as expected to both cell contexts Ψ and to the two kinds of variables x and u in variable contexts Γ.

Lemma 4.3.1 (Weakening, Exchange, Contraction)

1. (a) *If* $\Psi; \Gamma \vdash M : R$
   then $\Psi; (\Gamma, x{:}S) \vdash M : R$ *and* $\Psi; (\Gamma, u{:}S) \vdash M : R$.
   (b) *If* $\Psi; (\Gamma, x{:}S_1, y{:}S_2, \Gamma') \vdash M : R$
   then $\Psi; (\Gamma, y{:}S_2, x{:}S_1, \Gamma') \vdash M : R$.
   (c) *If* $\Psi; (\Gamma, u_1{:}S_1, u_2{:}S_2, \Gamma') \vdash M : R$
   then $\Psi; (\Gamma, u_2{:}S_2, u_1{:}S_1, \Gamma') \vdash M : R$.
   (d) $\Psi; (\Gamma, x{:}S_1, u{:}S_2, \Gamma') \vdash M : R$ *if and only if*
   $\Psi; (\Gamma, u{:}S_2, x{:}S_1, \Gamma') \vdash M : R$.

$$\frac{x{:}A \text{ in } \Gamma}{\Psi;\Gamma \vdash x : A} \qquad \frac{\Psi;\Gamma \vdash M : A \qquad \Psi;(\Gamma, x{:}A) \vdash N : B}{\Psi;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

$$\frac{\Psi;(\Gamma, x{:}A) \vdash M : B}{\Psi;\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Psi;\Gamma \vdash M : A \to B \quad \Psi;\Gamma \vdash N : A}{\Psi;\Gamma \vdash M\ N : B}$$

$$\frac{u{:}A \text{ in } \Gamma}{\Psi;\Gamma \vdash u{:}A} \qquad \frac{\Psi;(\Gamma, u{:}A) \vdash M : A}{\Psi;\Gamma \vdash \mathbf{fix}\ u.M : A}$$

$$\frac{l{:}A \text{ in } \Psi}{\Psi;\Gamma \vdash l : A\,\mathbf{ref}} \qquad \frac{\Psi;\Gamma \vdash M : A}{\Psi;\Gamma \vdash \mathbf{ref}\ M : A\,\mathbf{ref}} \qquad \frac{\Psi;\Gamma \vdash M : A\,\mathbf{ref}}{\Psi;\Gamma \vdash\ !\,M : A}$$

$$\frac{\Psi;\Gamma \vdash M : A\,\mathbf{ref} \quad \Psi;\Gamma \vdash N : A}{\Psi;\Gamma \vdash M := N : 1} \qquad \frac{}{\Psi;\Gamma \vdash () : 1}$$

$$\frac{\Psi;\Gamma \vdash V : A \qquad \Psi;\Gamma \vdash V : B}{\Psi;\Gamma \vdash V : A \,\&\, B} \qquad \frac{}{\Psi;\Gamma \vdash V : \top} \qquad \frac{\Psi;\Gamma \vdash M : A \qquad A \leq B}{\Psi;\Gamma \vdash M : B}$$

$$\frac{}{\Psi;\Gamma \vdash \epsilon : \mathsf{nat}} \qquad \frac{\Psi;\Gamma \vdash M : \mathsf{pos}}{\Psi;\Gamma \vdash M\ \mathsf{0} : \mathsf{pos}} \qquad \frac{\Psi;\Gamma \vdash M : \mathsf{bits}}{\Psi;\Gamma \vdash M\ \mathsf{0} : \mathsf{bits}}$$

$$\frac{\Psi;\Gamma \vdash M : \mathsf{nat}}{\Psi;\Gamma \vdash M\ \mathsf{1} : \mathsf{pos}} \qquad \frac{\Psi;\Gamma \vdash M : \mathsf{bits}}{\Psi;\Gamma \vdash M\ \mathsf{1} : \mathsf{bits}}$$

$$\frac{\Psi;\Gamma \vdash M : \mathsf{bits} \quad \Psi;\Gamma \vdash M_1 : A \quad \Psi;(\Gamma, x{:}\mathsf{bits}) \vdash M_2 : A \quad \Psi;(\Gamma, y{:}\mathsf{bits}) \vdash M_3 : A}{\Psi;\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ \mathsf{0} \Rightarrow M_2 \mid y\ \mathsf{1} \Rightarrow M_3 : A}$$

$$\frac{\Psi;\Gamma \vdash M : \mathsf{nat} \quad \Psi;\Gamma \vdash M_1 : A \quad \Psi;(\Gamma, x{:}\mathsf{pos}) \vdash M_2 : A \quad \Psi;(\Gamma, y{:}\mathsf{nat}) \vdash M_3 : A}{\Psi;\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ \mathsf{0} \Rightarrow M_2 \mid y\ \mathsf{1} \Rightarrow M_3 : A}$$

$$\frac{\Psi;\Gamma \vdash M : \mathsf{pos} \quad \Psi;(\Gamma, x{:}\mathsf{pos}) \vdash M_2 : A \quad \Psi;(\Gamma, y{:}\mathsf{nat}) \vdash M_3 : A}{\Psi;\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ \mathsf{0} \Rightarrow M_2 \mid y\ \mathsf{1} \Rightarrow M_3 : A}$$

Figure 4.1: Typing Rules

(e) *If* $\Psi; (\Gamma, x{:}S, y{:}S, \Gamma) \vdash M : R$
then $\Psi; (\Gamma, w{:}S, \Gamma') \vdash \{w/x\}\{w/y\}M : R$.

(f) *If* $\Psi; (\Gamma, u_1{:}S, u_2{:}S, \Gamma) \vdash M : R$
then $\Psi; (\Gamma, u_3{:}S, \Gamma') \vdash \{u_3/u_1\}\{u_3/u_2\}M : R$.

2. (a) *If* $\Psi; \Gamma \vdash M : R$
then $(\Psi, x{:}S); \Gamma \vdash M : R$.

(b) *If* $(\Psi, x{:}S_1, y{:}S_2, \Psi'); \Gamma \vdash M : R$
then $(\Psi, y{:}S_2, x{:}S_1, \Psi'); \Gamma \vdash M : R$.

(c) *If* $(\Psi, x{:}S, y{:}S, \Psi'); \Gamma \vdash M : R$
then $(\Psi, w{:}S, \Psi'); \Gamma \vdash \{w/x\}\{w/y\}M : R$.

Proof: By straightforward inductions over the structure of the derivations (as in Chapter 3). □

The value preservation lemma extends as expected. We introduce a second part to the lemma for expression variables u.

Lemma 4.3.2 (Value Preservation)

1. $\{V'/x\}V$ *is a value.*
2. $\{M/u\}V$ *is a value.*

Proof: By straightforward inductions on V. □

The substitution lemma extends as expected to value variables x.

Lemma 4.3.3 (Value Substitution Lemma)
If $\Psi; \Gamma \vdash V : A$ *and* $\Psi; (\Gamma, x{:}A) \vdash N : B$
then $\Psi; \Gamma \vdash \{V/x\}N : B$.

Proof: By a straightforward induction on the typing derivation for N (as in Chapter 3). □

We have an additional substitution lemma for expression variables u.

Lemma 4.3.4 (Expression Substitution Lemma)
If $\Psi; \Gamma \vdash M : A$ *and* $\Psi; (\Gamma, u{:}A) \vdash N : B$
then $\Psi; \Gamma \vdash \{M/u\}N : B$.

Proof: By induction on the typing derivation $\mathcal{D}_2$ for N. We show one interesting case. The remaining cases are straightforward and follow the previous substitution lemma.

Case: $\mathcal{D}_2 = \dfrac{\begin{array}{cc}\mathcal{D}_{21} & \mathcal{D}_{22} \\ \Psi; (\Gamma, u{:}R) \vdash V : S_1 & \Psi; (\Gamma, u{:}R) \vdash V : S_2\end{array}}{\Psi; (\Gamma, u{:}R) \vdash V : S_1 \,\&\, S_2}$

Applying the induction hypothesis to $\mathcal{D}_{21}$ and $\mathcal{D}_{22}$ yields derivations:

$\mathcal{D}_{31} :: \Psi \vdash_\Sigma \{M/u\}V : S_1$ and $\mathcal{D}_{32} :: \Psi \vdash_\Sigma \{M/u\}V : S_2$.
Since $\{M/u\}V$ is a value (by the second part of the value preservation lemma above) we can apply the intersection introduction rule to these derivations to obtain $\mathcal{D}_3 :: \Psi \vdash_\Sigma \{M/u\}V : S_1 \,\&\, S_2$, as required.

□

4.4 Typing of stores and states

Stores are typed using the following judgment.

$\Psi \vdash C : \Psi'$ Store C satisfies cell context Ψ'
when checked against cell context Ψ.

The rules for this judgment simply require each value to have the appropriate type under the empty variable context.

$$\frac{}{\Psi \vdash \cdot : \cdot} \qquad \frac{\Psi \vdash C' : \Psi' \qquad \Psi; \cdot \vdash V : A}{\Psi \vdash (C', l = V) : (\Psi', l{:}A)}$$

The following judgment defines typing of program states.

$\vdash (C \triangleright M) : (\Psi \triangleright A)$ Program state $(C \triangleright M)$ types with
cell context Ψ and type A.

It is defined directly from the previous typing judgments. We require the store C to satisfy the cell context Ψ under the same cell context. This allows consistent occurrences of cells l in the values in a cell context.

$$\frac{\Psi \vdash C : \Psi \qquad \Psi; \cdot \vdash M : A}{\vdash (C \triangleright M) : (\Psi \triangleright A)}$$

4.5 Reduction semantics

We now present a reduction style semantics for our language, roughly following Wright and Felleisen [WF94]. We start by defining *evaluation contexts*, namely expressions with a hole $[\,]$ within which a reduction may occur:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\,M \mid V\,E \\ & & \mid \mathbf{let}\ x = E\ \mathbf{in}\ M \\ & & \mid \mathbf{ref}\,E \mid \mathbf{!}\,E \mid E \mathbf{:=} M \mid V \mathbf{:=} E \\ & & \mid E\ \mathsf{0} \mid E\ \mathsf{1} \\ & & \mid \mathbf{case}\ E\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ \mathsf{0} \Rightarrow M_2 \mid y\ \mathsf{1} \Rightarrow M_3 \end{array}$$

We write $E[M]$ to indicate the term obtained by replacing the hole $[\,]$ in E by M.

$$
\begin{array}{rcl}
C \triangleright E[(\lambda x.M)\, V] & \mapsto & C \triangleright E[\{V/x\}M] \\
C \triangleright E[\mathbf{let}\ x = V\ \mathbf{in}\ M] & \mapsto & C \triangleright E[\{V/x\}M] \\
C \triangleright E[\mathbf{fix}\ u.M] & \mapsto & C \triangleright E[\{\mathbf{fix}\ u.M/u\}M] \\
\\
C \triangleright E[(\mathbf{ref}\, V)] & \mapsto & (C, l = V) \triangleright E[l] \\
 & & \qquad (l \text{ not in } C, E) \\
\\
(C_1, l = V, C_2) \triangleright E[!\, l] & \mapsto & (C_1, l = V, C_2) \triangleright E[V] \\
(C_1, l = V_1, C_2) \triangleright E[l := V_2] & \mapsto & (C_1, l = V_2, C_2) \triangleright E[()] \\
\\
C \triangleright E[\mathbf{case}\ \epsilon \quad \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow M_2 \mid y\ 1 \Rightarrow M_3] & \mapsto & C \triangleright E[M_1] \\
C \triangleright E[\mathbf{case}\ V\ 0\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow M_2 \mid y\ 1 \Rightarrow M_3] & \mapsto & C \triangleright E[\{V/x\}M_2] \\
C \triangleright E[\mathbf{case}\ V\ 1\ \mathbf{of}\ \epsilon \Rightarrow M_1 \mid x\ 0 \Rightarrow M_2 \mid y\ 1 \Rightarrow M_3] & \mapsto & C \triangleright E[\{V/y\}M_3]
\end{array}
$$

Figure 4.2: Reduction Rules

We write $C \triangleright M \mapsto C' \triangleright M'$ for a one-step computation, defined by the reduction rules in Figure 4.2. Each rule reduces a redex N that appears in an evaluation position in the term M, i.e. $M = E[N]$ for some E. We maintain the invariant that M does not contain free variables x or u and that all cells l in M are defined in C.

Critical in the proof of progress are the following inversion properties, similar to Lemma 3.8.5 (Inversion for λ-Abstractions) which was needed in the proof of subject reduction in Section 3.8. Also similar to that section, the proofs make use of the equivalence between declarative and algorithmic subtyping to reduce the number of cases that need to be considered.

These properties are generalizations of simpler properties in languages without subtyping, intersections, or effects.

Lemma 4.5.1 (Value Inversion)

1. *If* $\Psi; \cdot \vdash V : A$ *and* $A \trianglelefteq B_1 \to B_2$
   then $V = \lambda x.M$ *and* $\Psi; (x{:}B_1) \vdash M : B_2$.
2. *If* $\Psi; \cdot \vdash V : A$ *and* $A \trianglelefteq B$ **ref**
   then $V = l$ *and there is some* B' *such that* $l{:}B'$ *is in* Ψ, *and* $B' \trianglelefteq B$, *and* $B \trianglelefteq B'$.
3. *If* $\Psi; \cdot \vdash V : A$ *and* $A \trianglelefteq \mathsf{bits}$ *then we have one of the following cases:*
   (a) $V = \epsilon$
   (b) $V = (V_0\ 0)$ *and* $\Psi; \cdot \vdash V_0 : \mathsf{bits}$
   (c) $V = (V_1\ 1)$ *and* $\Psi; \cdot \vdash V_1 : \mathsf{bits}$
4. *If* $\Psi; \cdot \vdash V : A$ *and* $A \trianglelefteq \mathsf{nat}$ *then we have one of the following cases:*
   (a) $V = \epsilon$
   (b) $V = (V_0\ 0)$ *and* $\Psi; \cdot \vdash V_0 : \mathsf{pos}$

(c) $V = (V_1\ 1)$ *and* $\Psi; \cdot \vdash V_1 : \mathsf{nat}$

5. *If* $\Psi; \cdot \vdash V : A$ *and* $A \trianglelefteq \mathsf{pos}$ *then we have one of the following cases:*

   (a) $V = (V_0\ 0)$ *and* $\Psi; \cdot \vdash V_0 : \mathsf{pos}$

   (b) $V = (V_1\ 1)$ *and* $\Psi; \cdot \vdash V_1 : \mathsf{nat}$

Proof:
Each property is stated at a level of generality that allows it to be proved directly by inducting on the given typing derivation. For each we have inductive cases when the typing rule is subsumption or intersection introduction. The cases for top introduction cannot occur, by inversion on the assumed algorithmic subtyping derivation. The remaining cases are the introduction rules for the corresponding type constructors, and are straightforward. □

We are now ready to prove our main theorem, namely that our type system with mutable references and value-restricted intersections satisfies progress and type preservation, i.e., that programs can't go wrong as in the example in the introduction.

Theorem 4.5.2 (Progress and Type Preservation)
If $\vdash (C \triangleright M) : (\Psi \triangleright A)$ *then either*

1. M *is a value.*
2. $(C \triangleright M) \mapsto (C' \triangleright M')$ *for some* C', M' *and* Ψ' *satisfying* $\vdash (C' \triangleright M') : (\Psi, \Psi' \triangleright A)$.

Proof: By induction on the typing derivation for M.

- The case for subsumption is immediate, using the induction hypothesis.
- The cases for intersection introduction and top introduction are trivial: the value restriction forces M to be a value.
- For the remaining cases the typing rule matches the top term constructor of M.
- The cases for the typing rules corresponding to $\lambda x.M$, l, $()$ and ϵ are trivial, since they are values.
- The case for the typing rule corresponding to **fix** is easy, using the Expression Substitution Lemma (Lemma 4.3.4) to construct the required typing derivation.
- In the other cases, we apply the induction hypothesis to the subderivations for appropriate immediate subterms N_i of M which are in evaluation positions i.e. $M = E[N_i]$ (in each case, there is at least one).
  - If for some N_i the induction hypothesis yields $(C \triangleright N_i) \mapsto (C' \triangleright N_i')$ with $(C' \triangleright N_i') : (\Psi, \Psi' \triangleright B)$ then we can construct the required reduction and typing derivation for M.

– Otherwise, each immediate subterm N_i with $M = E[N_i]$ is a value. For these cases we apply the appropriate clause of the preceding inversion lemma, using reflexivity of algorithmic subtyping. In each case we find that M can be reduced to some M' and that we can construct the required typing for M', using the substitution lemma in some cases.

□

All of our restrictions are needed in this proof:

- The case of $E[!\, l]$ requires subtyping for A **ref** to be co-variant.
- The case of $E[l := V]$ requires subtyping for A **ref** to be contra-variant. With the previous point it means it must be non-variant.
- The value restriction for top is needed because otherwise the case for that rule would involve an arbitrary term M with no derivation to apply the induction hypothesis to.
- The value restriction for intersections is needed because otherwise the induction hypothesis is applied to the premises of the intersection introduction rule

$$\frac{\Psi; \cdot \vdash M : A_1 \qquad \Psi; \cdot \vdash M : A_2}{\Psi; \cdot \vdash M : A_1 \,\&\, A_2}$$

  which yields that for some C_1, M_1 and Ψ_1

$$(C \triangleright M) \mapsto (C_1 \triangleright M_1) \text{ and } \vdash (C_1 \triangleright M_1) : (\Psi, \Psi_1 \triangleright A_1)$$

  and also that for some C_2, M_2 and Ψ_2

$$(C \triangleright M) \mapsto (C_2 \triangleright M_2) \text{ and } \vdash (C_2 \triangleright M_2) : (\Psi, \Psi_2 \triangleright A_2)$$

  Even if we show that evaluation is deterministic (which shows $M_1 = M_2 = M'$ and $C_1 = C_2 = C'$), we have no way to reconcile Ψ_1 and Ψ_2 to a Ψ' such that

$$\vdash (C' \triangleright M') : (\Psi, \Psi' \triangleright A_1 \,\&\, A_2)$$

  because a new cell allocated in C_1 and C_2 may be assigned a different type in Ψ_1 and Ψ_2. It is precisely this observation which gives rise to the first counterexample in the introduction to Chapter 3 (and repeated at the start of this chapter).
- The absence of the distributivity rules is critical in the inversion property for values $V : A$ for $A \trianglelefteq B_1 \to B_2$ which relies on the property that if $A_1 \,\&\, A_2 \trianglelefteq B_1 \to B_2$ then either $A_1 \trianglelefteq B_1 \to B_2$ or $A_2 \trianglelefteq B_1 \to B_2$.

The analysis above indicates that if we fix the cells in the store and disallow new allocations by removing the **ref** M construct, the language would be sound even without a value restriction on intersection introduction as long as the **ref** type constructor is non-variant. Of course, real languages such as Standard ML do allow allocations, so in this case the value restriction is required.

Overall, this proof is not much more difficult than the case without intersection types, but this is partially because we have set up our definitions very carefully.

We omit the presentation of a bidirectional type-checking algorithm for $ML^{\&\mathbf{ref}}$, since our main purpose in this chapter was to demonstrate that our restrictions result in a form of intersection types that are suitable for extension with call-by-value effects. See [DP00] for a bidirectional type-checking algorithm for a similar language.

Chapter 5

Datasort declarations

The development in Chapter 3 was parameterized by a set of finite lattices of refinements for each type. In this chapter we focus on a particular mechanism for defining such lattices of refinements. This mechanism is based on the `rectype` declarations of Freeman and Pfenning [FP91, Fre94] which allows refinements of ML datatypes to be defined using a set of mutually recursive grammars.

We differ slightly from Freeman and Pfenning in that we do not allow nested applications of constructors in our definitions. We do this to avoid the anonymous refinements that result in their system, which result in error messages being less informative in some instances. As a result our refinement declarations more closely mirror the syntax of ML datatype declarations. Thus, we refer to these declarations as *datasort* declarations.

Examples of datasort declarations for refinements of a datatype for bit strings appeared in Section 1.4.3. These were also the basis for the example types in Chapter 4. We repeat these declarations here.

```
datatype bits = bnil | b0 of bits | b1 of bits

(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat ]*)
```

A datasort declaration is required to mirror the datatype declaration that it refines, except that some value constructors may be omitted, and where there is a type in the datatype declaration the datasort declaration may have any refinement of that type. Value constructors may also be repeated with different sorts for the constructor argument, as illustrated by the following example, repeated from Section 1.6.1.

```
(*[ datasort zero = bnil
         and nat2 = bnil | b0 of pos
                         | b1 of zero | b1 of pos  ]*)
```

This mechanism is certainly not the only interesting way to define refinements. For example, we might also consider refinements corresponding to subsets of integers, or refinements which capture the presence or absence of effects when an expression is executed. We have chosen to focus on recursive refinements of datatypes because datatypes play a key role in most ML

programs: conditional control flow is generally achieved via pattern matching with datatypes. Further, experience so far demonstrates that these refinements allow many common properties to be specified in real programs.

The main technical contribution presented in this chapter is an algorithm for checking inclusion between datasort declarations that is complete with respect to a simple inductive semantics for declarations with no occurrences of function types. This provides a simple and natural description of which inclusions a programmer should expect to hold, which was lacking in previous work on refinement types [Fre94]. We show how to extend this algorithm to declarations that include functions, including recursion in negative positions. This seems necessary in practice, but alas we cannot extend our inductive semantics to these declarations. We argue that our extension is still sensible, and our experience indicates that the inclusions determined by this algorithm are very natural.

Without occurrences of function types, datasort declarations are essentially a form of regular tree grammars for defining subtypes of datatypes. Similar types based on regular tree grammars have been considered by a number of researchers in logic programming, starting with Mishra [Mis84], and in functional programming by Aiken and Murphy [AM91], Hosoya, Vouillon and Pierce [HVP00], and Benzaken, Castagna, and Frisch [BCF03]. See Section 1.6.8 for more details. The latter three include algorithms for similar inclusion problems to ours. Our algorithm is particularly similar to the algorithm proposed by Hosoya, Vouillon and Pierce, although ours includes intersections. Both our algorithm and that of that of Hosoya, Vouillon and Pierce broadly follow the algorithm by Aiken and Murphy.

5.1 Syntax

As in Chapter 2 and Chapter 3, our language is parameterized by a set of base types and base sorts. However, in this chapter we require that the base types and base sorts be defined by datatype and datasort declarations in the signature. A datatype declaration has a body with a sequence of alternatives, each with a unique constructor applied to type. Constructors must belong to a unique datatype. For now, we restrict the types allowed in datatype definitions to *regular types*, those which do not contain function types, although we relax this restriction at the end of the chapter. A datasort definition defines a refinement of a particular datatype. The body of a datasort definition must be compatible with the definition of the datatype it refines. Each alternative in the datasort body must consist of one of the datatype's constructors applied to a refinement of the corresponding type, and constructors may be repeated in a datasort bodies.

We include here a language of terms inhabiting regular types. This language does not include functions, and hence also does not include variables bound by functions. As a result, these terms lack any interesting notion of reduction: without variables bound by functions, elimination forms can only be applied following the corresponding introduction forms, and there is no real point in constructing such terms. We have thus omitted elimination forms for terms in this chapter, so that all terms are values. The main purpose of this language of terms is to allow the definition of a semantics of sorts as sets of values, hence non-values would not be of much interest anyway.

$$
\begin{array}{rrcl}
\text{Regular Types} & A, B & ::= & a \mid 1 \mid A \times B \\
\text{Regular Sorts} & R, S & ::= & r \mid 1 \mid R \times S \mid R \,\&\, S \mid \top^A \\
\text{Datatype Bodies} & D & ::= & \cdot \mid D \sqcup c\,A \\
\text{Datasort Bodies} & \Psi & ::= & \cdot \mid \Psi \sqcup c\,R \\
\text{Signatures} & \Sigma & ::= & \Sigma, (a = D) \mid \Sigma, (r \stackrel{a}{=} \Psi) \\
\text{Terms (Values)} & M, N & ::= & c\,M \mid () \mid (M,N)
\end{array}
$$

We will also use V for terms when we want to emphasize that they are values, even though all terms are values in the current language. In what follows we generally omit the leading "$\cdot$" from non-empty datatype and datasort bodies. We also generally omit the datatype a from datasort declarations $r \stackrel{a}{=} \Psi$ when Ψ is non-empty, since then a is determined by the constructors that appear in Ψ. Interestingly, the sort 1 is equivalent to $\top^1$ in our semantics, so it could be omitted. We have chosen to retain it for consistency with the constructs for types.

5.2 Comparison with previous signatures

We now briefly compare the signatures in this chapter with those in previous chapters. The following signature corresponds to the datatype and datasort declarations for bit strings at the start of this chapter.

$$
\begin{array}{rrcl}
\boldsymbol{\Sigma}_{\mathsf{list}} = (\ a_{\mathsf{bits}} & = & c_{\mathsf{bnil}}\,1 \sqcup c_{\mathsf{b0}}\,a_{\mathsf{bits}} \sqcup c_{\mathsf{b1}}\,a_{\mathsf{bits}}, \\
r_{\mathsf{nat}} & = & c_{\mathsf{bnil}}\,1 \sqcup c_{\mathsf{b0}}\,a_{\mathsf{pos}} \sqcup c_{\mathsf{b1}}\,a_{\mathsf{nat}}, \\
r_{\mathsf{pos}} & = & c_{\mathsf{b0}}\,a_{\mathsf{pos}} \sqcup c_{\mathsf{b1}}\,a_{\mathsf{nat}}\ )
\end{array}
$$

This can be thought of as roughly similar to the following signature in the style of Chapter 2. (A corresponding signature in style of Chapter 3 can be obtained roughly by omitting the declarations of a_{bits}, r_{nat} and r_{pos}.)

$$
\begin{array}{rcll}
\boldsymbol{\Sigma}_{\mathsf{oldlist}} = (\ a_{\mathsf{bits}} & : & \mathsf{type}, & \\
r_{\mathsf{nat}} & \sqsubset & a_{\mathsf{bits}}, & \\
r_{\mathsf{pos}} & \sqsubset & a_{\mathsf{bits}}, & \\
c_{\mathsf{bnil}} & : & a_{\mathsf{bits}}, & c_{\mathsf{bnil}} \in r_{\mathsf{nat}}, \\
c_{\mathsf{b0}} & : & a_{\mathsf{bits}} \to a_{\mathsf{bits}}, & c_{\mathsf{b0}} \in (r_{\mathsf{pos}} \to r_{\mathsf{nat}}) \,\&\, (r_{\mathsf{pos}} \to r_{\mathsf{pos}}), \\
c_{\mathsf{b1}} & : & a_{\mathsf{bits}} \to a_{\mathsf{bits}}, & c_{\mathsf{b1}} \in (r_{\mathsf{nat}} \to r_{\mathsf{nat}}) \,\&\, (r_{\mathsf{nat}} \to r_{\mathsf{pos}})\ )
\end{array}
$$

However, the signatures in this chapter have an explicit "closed world assumption" which was absent in prior chapters. For example, the signature $\boldsymbol{\Sigma}_{\mathsf{list}}$ above explicitly rules out the possibility of the signature later being extended by adding a new constructor to the type a_{bits} and the refinements r_{nat} and r_{pos}: doing so might invalidate some inferences made prior to the extension. This is reflected by the use of "=" in the signature. In contrast the signatures in prior chapters were intended to be more open ended, and can be extended with new constructors without invalidating prior inferences (and thus could serve as a suitable foundation for sorts for languages quite different from ML, such as LF).

A consequence of this is that from the signature Σ_{list} we can conclude that the inclusion $r_{\mathsf{pos}} \leq r_{\mathsf{nat}}$ holds, but we cannot reach the same conclusion from signature $\Sigma_{\mathsf{oldlist}}$. This is why subsorting declarations were included in the signatures in prior chapters, but are not included in the signatures of this chapter: instead we will determine the inclusions by comparing the datasort bodies.

5.3 Validity judgments

The following six judgments are required to determine the validity of signatures. To allow mutual recursion in datatype and datasort declarations, each declaration in a signature must be judged valid with respect to the whole signature. Thus, the top-level validity judgment for signatures checks that a signature is valid with respect to itself via a judgment that checks each declaration in a signature. We also have a validity judgment for datatype bodies, and one for datasort bodies refining a datatype body. Additionally, we have judgments like those in prior chapters for validity of types and valid refinements of types.

$\vdash \Sigma\ \mathit{Sig}$	Σ is a valid signature
$\vdash_\Sigma \Sigma'\ \mathit{Sig}$	Σ' contains valid declarations under Σ
$\vdash_\Sigma D\ \mathit{Dat}$	D is a valid datatype body under Σ
$\vdash_\Sigma \Psi \sqsubset D$	Ψ is a valid datasort body refining D under Σ
$\vdash_\Sigma A : \mathsf{type}$	A is a valid type
$\vdash_\Sigma R \sqsubset A$	R is a valid refinement of type A.

Valid signatures

$$\frac{\vdash_\Sigma \Sigma\ \mathit{Sig}}{\vdash \Sigma\ \mathit{Sig}}\ \mathsf{sigsig}$$

Valid signature declarations

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{Sig}}\ \mathsf{sigemp} \qquad \frac{\vdash_\Sigma \Sigma'\ \mathit{Sig} \qquad \vdash_\Sigma D\ \mathit{Dat}}{\vdash_\Sigma (\Sigma', a = D)\ \mathit{Sig}}\ \mathsf{sigtyp}$$

$$\frac{\vdash_\Sigma \Sigma'\ \mathit{Sig} \qquad a = D \text{ in } \Sigma \qquad \vdash_\Sigma \Psi \sqsubset D}{\vdash_\Sigma (\Sigma', r \stackrel{a}{=} \Psi)\ \mathit{Sig}}\ \mathsf{sigsrt}$$

Valid datatype bodies

$$\frac{}{\vdash_\Sigma \cdot\ Dat}\ \textsf{datemp} \qquad \frac{\vdash_\Sigma D\ Dat \qquad c \text{ not in } D \qquad \vdash_\Sigma A : \textsf{type}}{\vdash_\Sigma (D \sqcup c\,A)\ Dat}\ \textsf{datcon}$$

Valid datasort bodies

$$\frac{}{\vdash_\Sigma \cdot \sqsubset D}\ \textsf{dsrtemp} \qquad \frac{\vdash_\Sigma \Psi \sqsubset D \qquad cA \text{ in } D \qquad \vdash_\Sigma R \sqsubset A}{\vdash_\Sigma (\Psi \sqcup cR) \sqsubset D}\ \textsf{dsrtcon}$$

Valid types

$$\frac{(a = D) \text{ in } \Sigma}{\vdash_\Sigma a : \textsf{type}}\ \textsf{typcon} \qquad \frac{}{\vdash_\Sigma 1 : \textsf{type}}\ \textsf{typunit} \qquad \frac{\vdash_\Sigma A : \textsf{type} \qquad \vdash_\Sigma B : \textsf{type}}{\vdash_\Sigma A \times B : \textsf{type}}\ \textsf{typprod}$$

Valid refinements

$$\frac{(r \stackrel{a}{=} \Psi) \text{ in } \Sigma}{\vdash_\Sigma r \sqsubset a}\ \textsf{srtcon} \qquad \frac{}{\vdash_\Sigma 1 \sqsubset 1}\ \textsf{srtunit} \qquad \frac{\vdash_\Sigma R \sqsubset A \qquad \vdash_\Sigma S \sqsubset B}{\vdash_\Sigma R \times S \sqsubset A \times B}\ \textsf{srtcross}$$

$$\frac{\vdash_\Sigma R \sqsubset A \qquad \vdash_\Sigma S \sqsubset A}{\vdash_\Sigma R \,\&\, S \sqsubset A}\ \textsf{srtinter} \qquad \frac{}{\vdash_\Sigma \top^A \sqsubset A}\ \textsf{srttop}$$

Finally, we have a validity judgment for terms, which does not require a context, since we have no functions or variables, and is particularly simple: we only have introduction rules for each construct. The rule for constructors uses the signature to check the argument type of the constructor, and the base type to which the constructor belongs.

Valid terms (values)

$\vdash_\Sigma M : A$ $\quad M$ is a valid term of type A (where A is a valid type)

$$\frac{(a = D) \text{ in } \Sigma \qquad cB \text{ in } D \qquad \vdash_\Sigma M : B}{\vdash_\Sigma c\,M : a}\ \textsf{objcon}$$

$$\frac{}{\vdash_\Sigma () : 1}\ \textsf{objunit} \qquad \frac{\vdash_\Sigma M : A \qquad \vdash_\Sigma N : B}{\vdash_\Sigma (M,N) : A \times B}\ \textsf{objprod}$$

5.4 Semantics of datasorts

We define the semantics of sorts and datasort bodies as subsets of the set of values with the appropriate type. Their definitions are inductive on the structure of terms: whether M is included in a sort only depends on inclusions in sorts of immediate subterms of M. The semantics of sorts also involves a subinduction on the structure of sorts for intersections, and on the structure of datasort bodies for $\sqcup$. The semantics specifies which terms are contained in the semantics of each sort, as follows.

Semantics of sorts

$$\begin{aligned}
M \in [\![r]\!] &\quad \text{if } M \in [\![\Psi]\!] \qquad \text{when } r = \Psi \text{ is in } \Sigma \\
() \in [\![1]\!] &\quad \text{(i.e. } [\![1]\!] = \{()\}\text{)} \\
(M, N) \in [\![R \times S]\!] &\quad \text{if } M \in [\![R]\!] \text{ and } N \in [\![S]\!] \\
M \in [\![R \,\&\, S]\!] &\quad \text{if } M \in [\![R]\!] \text{ and } M \in [\![S]\!] \\
M \in [\![\top^A]\!] &\quad \text{if } \vdash_\Sigma M : A
\end{aligned}$$

Semantics of datasort bodies

$$\begin{aligned}
M \notin [\![\cdot]\!] &\quad \text{(i.e. } [\![\cdot]\!] = \{\}\text{)} \\
M \in [\![\Psi \sqcup c\,R]\!] &\quad \text{if } M \text{ in } [\![\Psi]\!] \text{ or } (M = c\,N \text{ and } N \in [\![R]\!])
\end{aligned}$$

We can reformulate this semantics to directly specify a set for each sort, rather than specifying which elements are included in the semantics of each sort. However, doing so hides the inductive nature of the definition: $[\![r]\!]$ is defined in terms of $[\![\Psi]\!]$ which might in turn be defined in terms of $[\![r]\!]$, thus the definition appears to be circular. Thus, the definition above is preferable, since it makes it clear that the semantics is inductive on the structure of values. Regardless, the reformulated semantics follows, mostly for comparison.

Reformulated semantics of sorts

$$\begin{aligned}
[\![r]\!] &= [\![\Psi]\!] \qquad \text{(when } r = \Psi \text{ is in } \Sigma\text{)} \\
[\![1]\!] &= \{()\} \\
[\![R \times S]\!] &= \{(M, N) \mid M \in [\![R]\!], N \in [\![S]\!]\} \\
[\![R \,\&\, S]\!] &= [\![R]\!] \cap [\![S]\!] \\
[\![\top^A]\!] &= \{M \mid \vdash_\Sigma M : A\}
\end{aligned}$$

Reformulated semantics of datasort bodies

$$\begin{aligned}
[\![\cdot]\!] &= \{\} \\
[\![\Psi \sqcup c\,R]\!] &= [\![\Psi]\!] \cup \{c\,M \mid M \in [\![R]\!]\}
\end{aligned}$$

We use the semantics of sorts to define subsorting via subset, in the obvious way.

Definition 5.4.1 *The subsorting $R \leq S$ holds exactly when $[\![R]\!] \subseteq [\![S]\!]$.*

5.5 Inclusion algorithm

We now present an algorithmic judgment which determines inclusion between two sorts. The algorithm requires a judgment that is generalized to determine inclusion of a sort R in a union of sorts U. This is because when we have datasort declarations like $r = c\,R$ and $s = c\,S_1 \sqcup c\,S_2$, if we want to check the inclusion $r \leq s$ we need to check whether R is contained in the union of S_1 and S_2. We thus introduce a syntax for "sort unions", which are a list of sorts, representing their union (see below), similar to datasort bodies only without constructors. The semantics of a union of sorts is defined in the obvious way: as the unions of sets obtained via the semantics of the sorts. Our algorithm is designed to produce the same result regardless of the order or replication of sorts in a union or datasort body, so we often treat them like sets. In particular we use the following notations to form sort unions and datasort bodies from finite sets.

$$\begin{aligned} \sqcup\{R_1, \ldots, R_n\} &= R_1 \sqcup \ldots \sqcup R_n \\ \sqcup\{c_1\,R_1, \ldots, c_n\,R_n\} &= c_1\,R_1 \sqcup \ldots \sqcup c_n\,R_n \end{aligned}$$

Our algorithm determines recursive inclusions by accumulating "assumptions" for subsorting goals which are currently "in progress". This technique has been used in a number of similar subtyping and equivalence algorithms for recursive types. Brandt and Henglein [BH98] have considered this technique in detail, and point out that it is essentially a way of inductively formulating a relation that is more naturally characterized coinductively.

In our case, we only accumulate such assumptions for refinements of base types, since this is sufficient to guarantee termination of the algorithm. This also leads to some advantages in the implementation: fewer and simpler checks against a smaller set of assumptions. To ensure termination, we require a finite bound on the number of potential assumptions. We do this by treating intersections of base sorts $r_1 \,\&\, \ldots \,\&\, r_n$ like sets $\{r_1, \ldots, r_n\}$. Since there are only a finite number of base sorts, there are only a finite number of such sets. Similarly, we treat unions of intersections of base sorts as sets of sets of base sorts, and there are only a finite number of these. Formally, we assume that we have a function that maps each intersection of base sorts ρ to a canonical representative $\lfloor\rho\rfloor$ which has the same set of base sorts as ρ, similarly a function which maps each base union u to a canonical representative $\lfloor u \rfloor$. In the implementation these unique representatives are sorted lists with respect to a total order based on the internal numerical identifiers for the base sorts.

$$\begin{array}{rrcl} \text{Sort unions} & U & ::= & \cdot \mid U \sqcup R \\ \text{Base refinements} & \rho & ::= & r \mid \rho_1 \,\&\, \rho_2 \mid \top^a \\ \text{Base unions} & u & ::= & \cdot \mid u \sqcup \rho \\ \text{Subsorting assumptions} & \Theta & ::= & \cdot \mid \Theta, (\rho \leq u) \end{array}$$

We now present our algorithm as a judgment with algorithmic rules, making use of three auxiliary judgments. These judgments will be explained in detail after the presentation of their rules, along with two required functions body and ubody.

In the rules that follow there will always be a single fixed signature Σ, which we often omit to avoid clutter.

$\Theta \vdash_\Sigma R \trianglelefteq U$	Sort R is included in sort union U under the assumptions Θ.
$\Theta \vdash_\Sigma \Psi_1 \trianglelefteq \Psi_2$	Datasort body Ψ_1 is included in datasort body Ψ_2 under the assumptions Θ.
$\Theta \vdash_\Sigma (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$	The product of R_1 minus U_1 and R_2 minus U_2 is contained in U, under the assumptions Θ.
$R \cong R_1 \otimes R_2$	Sort R is equivalent to the product of R_1 and R_2.

$$\frac{\lfloor\rho\rfloor \leq \lfloor u\rfloor \text{ in } \Theta}{\Theta \vdash \rho \trianglelefteq u} \qquad \frac{\lfloor\rho\rfloor \leq \lfloor u\rfloor \text{ not in } \Theta \qquad \Theta, \lfloor\rho\rfloor \leq \lfloor u\rfloor \vdash \mathsf{body}\lfloor\rho\rfloor \trianglelefteq \mathsf{ubody}\lfloor u\rfloor}{\Theta \vdash \rho \trianglelefteq u}$$

$$\frac{R \sqsubset 1}{\Theta \vdash R \trianglelefteq U \sqcup S} \qquad \frac{R \cong R_1 \otimes R_2 \qquad \Theta \vdash (R_1 \backslash \cdot) \otimes (R_2 \backslash \cdot) \trianglelefteq U}{\Theta \vdash R \trianglelefteq U}$$

$$\frac{}{\Theta \vdash \cdot \trianglelefteq \Psi} \qquad \frac{\Theta \vdash \Psi_1 \trianglelefteq \Psi_2 \qquad \Theta \vdash R \trianglelefteq \sqcup\{S \mid cS \text{ in } \Psi_2\}}{\Theta \vdash \Psi_1 \sqcup cR \trianglelefteq \Psi_2}$$

$$\frac{S \cong S_1 \otimes S_2 \qquad \begin{array}{l}\Theta \vdash (R_1 \backslash U_1 \sqcup S_1) \otimes (R_2 \backslash U_2) \trianglelefteq U \\ \Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2 \sqcup S_2) \trianglelefteq U\end{array}}{\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U \sqcup S}$$

$$\frac{\Theta \vdash R_1 \trianglelefteq U_1}{\Theta \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq \cdot} \qquad \frac{\Theta \vdash R_2 \trianglelefteq U_2}{\Theta \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq \cdot}$$

$$\frac{}{R_1 \times R_2 \cong R_1 \otimes R_2} \qquad \frac{R \cong R_1 \otimes R_2 \qquad S \cong S_1 \otimes S_2}{R \,\&\, S \cong (R_1 \& S_1) \otimes (R_2 \& S_2)} \qquad \frac{}{\top^{A \times B} \cong \top^A \otimes \top^B}$$

The main algorithmic subsorting judgment has two rules for the case when the sorts refine a base type. The first applies when the goal is equivalent to one of the assumptions, in which case we succeed. Otherwise, we use an auxiliary judgment to compare the bodies of the base sorts involved. This uses functions $\mathsf{body}_\Sigma(\rho)$ and $\mathsf{ubody}_\Sigma(u)$ which use the bodies for base sorts in Σ to construct datasort bodies for intersections ρ of base sorts and unions u of those intersections. The definitions these functions appear below, followed by a brief explanation.

$$
\begin{aligned}
\mathsf{body}(r) &= \Psi && \text{when } r = \Psi \text{ in } \Sigma \\
\mathsf{body}(\rho_1 \& \rho_2) &= \sqcup \{ c(R_1 \& R_2) \mid cR_1 \text{ in } \mathsf{body}(\rho_1), cR_2 \text{ in } \mathsf{body}(\rho_2) \} \\
\mathsf{body}(\top^a) &= \sqcup \{ c\top^A \mid cA \text{ in } D \} && \text{when } a = D \text{ in } \Sigma \\
\\
\mathsf{ubody}(\cdot) &= \cdot && \text{(empty datasort body)} \\
\mathsf{ubody}(u \sqcup \rho) &= \mathsf{ubody}(u) \sqcup \mathsf{body}(\rho)
\end{aligned}
$$

The body of an intersection an intersection $\rho_1 \& \rho_2$ is formed by intersecting the bodies of each part, which is achieved by intersecting all combinations of cR_1 and cR_2 from the respective bodies which have the same constructor c. There may be many combinations for a single c, since a datasort body may repeat a constructor: in this case the definition essentially distributes intersections into unions.

The auxiliary judgment for comparing datasort bodies checks that each component cR in the body on the left has R contained in the union of sorts S for the same constructor in the body on the right.

Returning to the main judgment for the algorithm, when the sorts refine the unit type $\mathsf{1}$ we succeed provided there is at least one sort in the union on the right hand side: there is only one refinement of the type $\mathsf{1}$, up to equivalence. We fail if we have a the goal like $\mathsf{1} \trianglelefteq \cdot$, and in fact this is the only point where failures originate in the algorithm: in every other case some rule applies.

When the sorts refine a product type $A \times B$ we use an auxiliary judgment which checks the inclusion of a product in a union of products. The idea here is that every refinement of a product type is equivalent to a product sort $R_1 \times R_2$: intersections are distributed inwards by the judgment $R \stackrel{\rightrightarrows}{\simeq} R_1 \otimes R_2$. Then, we can check whether $R_1 \times R_2$ is included in $U \sqcup (S_1 \times S_2)$ by subtracting $S_1 \times S_2$ from $R_1 \times R_2$ and checking whether this difference is contained in U. This difference $(R_1 \times R_2) \backslash (S_1 \times S_2)$ is equivalent to the union of $(R_1 \backslash S_1) \times R_2$ and $R_1 \times (R_2 \backslash S_2)$. This leads to the the first rule for the auxiliary judgment, which essentially checks that each part of this union is contained in U. The remaining two rules for the auxiliary judgment apply when the union U is empty, in which case one of the two components of the product on the left must be empty.

This approach to subtyping for products with unions is somewhat similar that taken by Hosoya, Vouillon and Pierce [HVP00] (and considered earlier by Aiken and Murphy [AM91]). One difference is that the form of our auxiliary judgment avoids the need for a rule with an explicit universal quantification over partitions of the union. It thus allows more "structural" proofs over the form of derivations.

5.6 Correctness of the inclusion algorithm

We now prove correctness of the inclusion algorithm with respect to the semantics of datasorts. We first prove that the algorithm always terminates, then we show soundness, and finally completeness.

Theorem 5.6.1 (Termination of $\trianglelefteq$)
There is no infinite sequence $J_1, J_2, \ldots$ with each J_n a conclusion of an instance of one of the algorithmic inclusion rules, and J_{n+1} one of the corresponding premises.

Proof: Each J_{n+1} is less than J_n with respect to the well order defined as follows.
First we define the (non-empty) set of types associated with J_n, as follows.

- The set of types associated with $\Theta \vdash R \trianglelefteq U$ is $\{A\}$ where $R \sqsubset A$.
- The set of types associated with $\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$ is $\{A_1, A_2\}$ where $R_1 \sqsubset A_1$ and $R_2 \sqsubset A_2$.
- The set of types associated with $\Theta \vdash \Psi_1 \trianglelefteq \Psi_2$ is $\{A_1, \ldots, A_n\}$ where $\Psi_1 = c_1\, R_1 \sqcup \ldots \sqcup c_n\, R_n$ and $R_1 \sqsubset A_1, \ldots, R_n \sqsubset A_n$.

Then, the well order is defined lexicographically on: the assumption set of J_n, followed by the types associated with J_n, followed by an order based on which judgment J_n is an instance of, followed by an ordering between instances of the same judgment based on the syntactic inclusion of unions in J_n. More precisely, the well-ordering is as follows.

- $J_{n+1} < J_n$ if J_{n+1} has a proper superset of the assumptions of J_n. (Note: for each signature Σ there are only a finite number of potential subsorting assumptions, hence the number of assumptions is bounded.)
- $J_{n+1} < J_n$ if J_{n+1} has the same assumptions as J_n and each type associated with J_{n+1} is syntactically less than (i.e., a proper subterm of) some type associated with J_n.
- $J_{n+1} < J_n$ if J_{n+1} has the same assumptions as J_n and each type associated with J_{n+1} is syntactically equal to some type associated with J_n and one of the following applies:
  1. (a) $J_{n+1} = \Theta \vdash R \trianglelefteq U$ and $J_n = \Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U_3$
     (b) $J_{n+1} = \Theta \vdash R \trianglelefteq U$ and $J_n = \Theta \vdash \Psi_1 \trianglelefteq \Psi_2$
  2. (a) $J_{n+1} = \Theta \vdash (R_1 \backslash U_{11}) \otimes (R_2 \backslash U_{12}) \trianglelefteq U_{13}$ and
     $J_n \quad = \Theta \vdash (R_1 \backslash U_{21}) \otimes (R_2 \backslash U_{22}) \trianglelefteq U_{23}$ and
     U_{13} is syntactically less than U_{23}.
     (b) $J_{n+1} = \Theta \vdash \Psi_{11} \trianglelefteq \Psi_2$ and $J_n = \Theta \vdash \Psi_{21} \trianglelefteq \Psi_2$ and Ψ_{11} is syntactically less than Ψ_{21}.

This is a well order because it is defined lexicographically from four well orders.

- The first order is a well order because the number of assumptions is bounded (given a particular signature).
- The second is a well ordering because every J_n has at least one type associated with it, hence an infinite decreasing sequence of sets of types associated with J_n would imply an infinite decreasing sequence of types (i.e. with each contained in the previous one).
- The third order is a well order because no decreasing sequence has length longer than one.

- The fourth order is a well order because it is directly based on a syntactic ordering.

It is straightforward to check that the premises of each rule are less than the conclusion according this well order.

□

The soundness and completeness proofs require the following lemmas.

Lemma 5.6.2 $[\![\mathsf{body}(\rho)]\!] = [\![\rho]\!]$ *and* $[\![\mathsf{ubody}(u)]\!] = [\![u]\!]$.

Proof: By induction on the structure of ρ and u. □

Lemma 5.6.3 *If every* r *in* ρ_1 *is in* ρ_2 *then* $[\![\rho_2]\!] \subseteq [\![\rho_1]\!]$.

Proof: By induction on the structure of ρ_1. □

Corollary 5.6.4 $[\![\lfloor\rho\rfloor]\!] = [\![\rho]\!]$.

Lemma 5.6.5 *If for every* ρ_1 *in* u_1 *there is* ρ_2 *in* u_2 *such that* $[\![\rho_1]\!] \subseteq [\![\rho_2]\!]$ *then* $[\![u_1]\!] \subseteq [\![u_2]\!]$.

Proof: By induction on the structure of u_1. □

Corollary 5.6.6 $[\![\lfloor u\rfloor]\!] = [\![u]\!]$.

Lemma 5.6.7 *If* $R \sqsubset 1$ *then* $[\![R]\!] = \{()\}$.

Proof: By induction on R. □

Lemma 5.6.8
If $R \sqsubset A_1 \times A_2$ *then* $R \stackrel{\rightrightarrows}{\cong} R_1 \otimes R_2$
for some unique $R_1 \sqsubset A_1$ *and unique* $R_2 \sqsubset A_2$ *with* $[\![R]\!] = [\![R_1 \times R_2]\!]$.

Proof: By induction on R. □

Lemma 5.6.9 (Datasort substitution)

1. *If* $\Theta \vdash \rho \trianglelefteq u$ *and* $\Theta, \rho{\leq}u \vdash R \trianglelefteq U$ *then* $\Theta \vdash R \trianglelefteq U$.
2. *If* $\Theta \vdash \rho \trianglelefteq u$ *and* $\Theta, \rho{\leq}u \vdash \Psi_1 \trianglelefteq \Psi_2$ *then* $\Theta \vdash \Psi_1 \trianglelefteq \Psi_2$.
3. *If* $\Theta \vdash \rho \trianglelefteq u$ *and* $\Theta, \rho{\leq}u \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$
   then $\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$.

Proof: By induction on the structure of the second derivation. All cases simply rebuild the derivation, except that for when the assumption $\rho \leq u$ is used, which follows.

Case: $\dfrac{\rho{\leq}u \text{ in } \Theta, \rho{\leq}u}{\Theta, \rho{\leq}u \vdash \rho \trianglelefteq u}$

Then $\Theta \vdash \rho \trianglelefteq u$, by the first assumption. □

For the soundness proof we need generalize the induction hypothesis to include an appropriate form for sort unions: recall that the semantics for sort unions is defined in the obvious way using set union.

Theorem 5.6.10 (Soundness of $\trianglelefteq$)

1. *If* $\cdot \vdash_\Sigma R \trianglelefteq U$ *and* $V \in [\![R]\!]$ *then* $V \in [\![U]\!]$.

2. *If* $\cdot \vdash_\Sigma \Psi_1 \trianglelefteq \Psi_2$ *and* $cV \in [\![\Psi_1]\!]$ *then* $cV \in [\![\Psi_2]\!]$.

3. *If* $\cdot \vdash_\Sigma (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$
   and $V_1 \in ([\![R_1]\!] \backslash [\![U_1]\!])$ *and* $V_2 \in ([\![R_2]\!] \backslash [\![U_2]\!])$
   and $V = (V_1, V_2)$ *then* $V \in [\![U]\!]$.

Proof: By induction on the structure of V and the inclusion derivation, lexicographically. We have the following cases for the derivation.

Case: $\dfrac{\lfloor\rho\rfloor \leq \lfloor u\rfloor \text{ in } \cdot}{\cdot \vdash \rho \trianglelefteq u}$

Cannot occur, since $\cdot$ is empty.
(Assumptions are recursively "unfolded" via substitution, which succeeds because the induction is principally on the structure of values.)

Case: $\dfrac{\lfloor\rho\rfloor \leq \lfloor u\rfloor \text{ not in } \cdot \qquad \lfloor\rho\rfloor \leq \lfloor u\rfloor \vdash \mathsf{body}\lfloor\rho\rfloor \trianglelefteq \mathsf{ubody}\lfloor u\rfloor}{\cdot \vdash \rho \trianglelefteq u}$

$\lfloor\rho\rfloor \leq \lfloor u\rfloor \vdash \mathsf{body}\lfloor\rho\rfloor \trianglelefteq \mathsf{ubody}\lfloor u\rfloor$	Assumed
$\lfloor\lfloor\rho\rfloor\rfloor \leq \lfloor\lfloor u\rfloor\rfloor \vdash \mathsf{body}\lfloor\lfloor\rho\rfloor\rfloor \trianglelefteq \mathsf{ubody}\lfloor\lfloor u\rfloor\rfloor$	$\lfloor\lfloor\rho\rfloor\rfloor = \lfloor\rho\rfloor$ and $\lfloor\lfloor u\rfloor\rfloor = \lfloor u\rfloor$
$\cdot \vdash \lfloor\rho\rfloor \trianglelefteq \lfloor u\rfloor$	Rule
$\cdot \vdash \mathsf{body}\lfloor\rho\rfloor \trianglelefteq \mathsf{ubody}\lfloor u\rfloor$.	Subst. Lemma (5.6.9)
$V \in [\![\rho]\!]$	Assumed
$V = cV'$ with $cV' \in [\![\rho]\!]$	Def. $[\![\rho]\!]$
$cV' \in [\![\mathsf{body}\lfloor\rho\rfloor]\!]$	Lemma 5.6.2, Corollary 5.6.4
$cV' \in [\![\mathsf{ubody}\lfloor u\rfloor]\!]$	Ind. Hyp. (V' is smaller)
$cV' \in [\![u]\!]$	Lemma 5.6.2, Corollary 5.6.6
$V \in [\![u]\!]$	Since $V = cV'$, above

Case: $\dfrac{R \sqsubset 1}{\cdot \vdash R \trianglelefteq U \sqcup S}$

$[\![R]\!] = \{()\}$	Lemma 5.6.7
$V = ()$	Since $V \in [\![R]\!]$ (assumed).
$S \sqsubset 1$	Well-formedness of $\cdot \vdash R \trianglelefteq U \sqcup S$
$[\![S]\!] = \{()\}$	Lemma 5.6.7
$V \in [\![S]\!]$	Since $V = ()$
$V \in [\![U \sqcup S]\!]$	Def. $[\![\cdot]\!]$

Case: $\dfrac{R \stackrel{\Rightarrow}{\simeq} R_1 \otimes R_2 \qquad \cdot \vdash (R_1 \backslash \cdot) \otimes (R_2 \backslash \cdot) \trianglelefteq U}{\cdot \vdash R \trianglelefteq U}$

$V \in [\![R]\!]$	Assumed
$V \in [\![R_1 \times R_2]\!]$	Lemma 5.6.8
$V = (V_1, V_2)$ with $V_1 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$	Def. $[\![\cdot]\!]$
$V \in [\![U]\!]$	Ind. Hyp. (same V, subderivation)

Case: $\dfrac{}{\cdot \vdash \cdot \trianglelefteq \Psi}$

Cannot occur, since $[\![\cdot]\!]$ is empty.

Case: $\dfrac{\cdot \vdash \Psi_1 \trianglelefteq \Psi_2 \qquad \cdot \vdash R \trianglelefteq \sqcup\{S \mid cS \text{ in } \Psi_2\}}{\cdot \vdash \Psi_1 \sqcup cR \trianglelefteq \Psi_2}$

$V \in [\![\Psi_1 \sqcup cR]\!]$	Assumed
$V \in [\![\Psi_1]\!]$ or $V \in \{cW \mid W \in [\![R]\!]\}$	Def. $[\![\cdot]\!]$
Suppose $V \in [\![\Psi_1]\!]$:	
$\quad V \in [\![\Psi_2]\!]$	Ind. Hyp. (same V, subderivation)
Otherwise:	
$\quad V = cW$ for some $W \in [\![R]\!]$	
$\quad W \in [\![\sqcup\{S \mid cS \text{ in } \Psi_2\}]\!]$	Ind. Hyp. (same V, subderivation)
$\quad W \in [\![S]\!]$ with cS in Ψ_2	Def. $[\![\cdot]\!]$
$\quad cW \in [\![\Psi_2]\!]$	Def. $[\![\cdot]\!]$

$$\frac{\begin{array}{ll} & \cdot \vdash (R_1\backslash U_1 \sqcup S_1) \otimes (R_2\backslash U_2) \trianglelefteq U \\ S \stackrel{\rightrightarrows}{\cong} S_1 \otimes S_2 & \cdot \vdash (R_1\backslash U_1) \otimes (R_2\backslash U_2 \sqcup S_2) \trianglelefteq U \end{array}}{\cdot \vdash (R_1\backslash U_1) \otimes (R_2\backslash U_2) \trianglelefteq U \sqcup S}$$

Case:

If $(V_1,V_2) \in [\![S]\!]$ then:

$(V_1,V_2) \in [\![U \sqcup S]\!]$ Def. $[\![\cdot]\!]$

Otherwise:

$(V_1,V_2) \notin [\![S_1 \times S_2]\!]$ Lemma 5.6.8

$V_1 \notin S_1$ or $V_2 \notin S_2$ Def. $[\![S_1 \times S_2]\!]$

Suppose $V_1 \notin S_1$ (the case for is $V_2 \notin S_2$ is dual)

$V_1 \in [\![R_1]\!]\backslash[\![U_1]\!]$ and $V_2 \in [\![R_2]\!]\backslash[\![U_2]\!]$ Assumed

$V_1 \in [\![R_1]\!]\backslash[\![U_1 \sqcup S_1]\!]$ Def. $[\![\cdot]\!]$

$(V_1,V_2) \in [\![U]\!]$ Ind. Hyp. (same V, subderivation)

Case: $\dfrac{\cdot \vdash R_1 \trianglelefteq U_1}{\cdot \vdash R_1\backslash U_1 \otimes R_2\backslash U_2 \trianglelefteq \cdot}$

$V_1 \in [\![R_1]\!]\backslash[\![U_1]\!]$ and $V_2 \in [\![R_2]\!]\backslash[\![U_2]\!]$ Assumed

$V_1 \in [\![R_1]\!]$ Def. $[\![R_1]\!]\backslash[\![U_1]\!]$

$V_1 \in [\![U_1]\!]$ Ind. Hyp. (V_1 smaller)

Contradicts $V_1 \in [\![R_1]\!]\backslash[\![U_1]\!]$.

Hence, case cannot occur.

Case: $\dfrac{\cdot \vdash R_2 \trianglelefteq U_2}{\cdot \vdash R_1\backslash U_1 \otimes R_2\backslash U_2 \trianglelefteq \cdot}$

Dual to the previous case.

□

To show the completeness of our algorithm, we first show that failures only occur when an inclusion should be rejected.

Theorem 5.6.11 (Soundness of Failure of $\trianglelefteq$)

1. *If $\Theta \vdash_\Sigma R \trianglelefteq U$ fails then there exists $V \in [\![R]\!]$ such that $V \notin [\![U]\!]$.*
2. *If $\Theta \vdash_\Sigma \Psi_1 \trianglelefteq \Psi_2$ fails*
   then there exists $cV \in [\![\Psi_1]\!]$ such that $cV \notin [\![\Psi_2]\!]$.
3. *If $\Theta \vdash_\Sigma (R_1\backslash U_1) \otimes (R_2\backslash U_2) \trianglelefteq U$ fails*
   then there exists $V_1 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$
   such that $V_1 \notin [\![U_1]\!]$ and $V_2 \notin [\![U_2]\!]$ and $(V_1,V_2) \notin [\![U]\!]$.

Proof: By induction on the termination well-ordering of the algorithm (from Theorem 5.6.1), i.e., following the order of the steps performed by the algorithm.

Case: $\Theta \vdash \rho \trianglelefteq u$ fails.

Then $\lfloor \rho \rfloor \leq \lfloor u \rfloor$ not in Θ
and $\Theta, \lfloor \rho \rfloor \leq \lfloor u \rfloor \vdash \mathsf{body}\lfloor \rho \rfloor \trianglelefteq \mathsf{ubody}\lfloor u \rfloor$ fails.
By ind. hyp., there is $cV \in [\![\mathsf{body}\lfloor \rho \rfloor]\!]$ such that $cV \notin [\![\mathsf{ubody}\lfloor u \rfloor]\!]$.
Then, $cV \in [\![\rho]\!]$ and $cV \notin [\![u]\!]$ (Lemma 5.6.2, Corollary 5.6.4 and Corollary 5.6.6).

Case: $\Theta \vdash R \trianglelefteq \cdot$ fails with $R \sqsubset 1$.

Then $() \in [\![R]\!]$ and $() \notin [\![\cdot]\!]$.

Case: $\Theta \vdash R \trianglelefteq U$ fails because $\Theta \vdash (R_1 \backslash \cdot) \otimes (R_2 \backslash \cdot) \trianglelefteq U$ fails with $R \stackrel{\rightrightarrows}{\simeq} R_1 \otimes R_2$.

Then, the ind. hyp. yields $V_1 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$ with $(V_1, V_2) \notin [\![U]\!]$.
And then, $(V_1, V_2) \in [\![R]\!]$ (by Lemma 5.6.8).

Case: $\Theta \vdash \Psi_1 \sqcup cR \trianglelefteq \Psi_2$ fails because $\Theta \vdash \Psi_1 \trianglelefteq \Psi_2$ fails.

Then, the ind. hyp. yields $V \in [\![\Psi_1]\!]$ with $V \notin [\![\Psi_2]\!]$.
And then, $V \in [\![\Psi_1 \sqcup cR]\!]$.

Case: $\Theta \vdash \Psi_1 \sqcup cR \trianglelefteq \Psi_2$ fails because $\Theta \vdash R \trianglelefteq \sqcup \{S \mid cS \text{ in } \Psi_2\}$ fails.

Then, the ind. hyp. yields $V \in [\![R]\!]$ with $V \notin [\![\sqcup \{S \mid cS \text{ in } \Psi_2\}]\!]$.
And then, $cV \in [\![cR]\!]$ with $cV \notin [\![\sqcup \{cS \mid cS \text{ in } \Psi_2\}]\!]$.
So, $cV \in [\![\Psi_1 \sqcup cR]\!]$ with $cV \notin [\![\Psi_2]\!]$.

Case: $\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U \sqcup S$ fails because
$\Theta \vdash (R_1 \backslash U_1 \sqcup S_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$ fails with $S \stackrel{\rightrightarrows}{\simeq} S_1 \otimes S_2$.

Then, the ind. hyp. yields $V_1 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$ with $V_1 \notin [\![U_1 \sqcup S_1]\!]$ and $V_2 \notin [\![U_2]\!]$ and $(V_1, V_2) \notin [\![U]\!]$.
And then, $V_1 \notin [\![U_1]\!]$.
Also, $V_1 \notin [\![S_1]\!]$ thus $(V_1, V_2) \notin [\![S_1 \times S_2]\!]$.
So, $(V_1, V_2) \notin [\![S]\!]$ (Lemma 5.6.8).
Thus, $(V_1, V_2) \notin [\![U \sqcup S]\!]$.

Case: $\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U \sqcup S$ fails because
$\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2 \sqcup S_2) \trianglelefteq U$ fails with $S \stackrel{\rightrightarrows}{\simeq} S_1 \otimes S_2$.

Dual to the previous case.

Case: $\Theta \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq \cdot$ fails
because $\Theta \vdash R_1 \trianglelefteq U_1$ fails and $\Theta \vdash R_2 \trianglelefteq U_2$ fails.

Then, the ind. hyp. yields $V_1 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$ with $V_1 \notin [\![U_1]\!]$ and $V_2 \notin [\![U_2]\!]$.
Further, $(V_1, V_2) \notin [\![\cdot]\!]$.

□

Now, to prove our completeness result, we combine the previous result with the termination theorem.

Theorem 5.6.12 (Completeness of $\trianglelefteq$)

1. *If* $[\![R]\!] \subseteq [\![U]\!]$ *then* $\cdot \vdash R \trianglelefteq U$.
2. *If* $[\![\Psi_1]\!] \subseteq [\![\Psi_2]\!]$ *then* $\cdot \vdash \Psi_1 \trianglelefteq \Psi_2$.
3. *If* $([\![R_1]\!] \backslash [\![U_1]\!]) \times ([\![R_2]\!] \backslash [\![U_2]\!]) \subseteq [\![U]\!]$ *then* $\cdot \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$.

Proof: By contradiction. We show only the case for $R \trianglelefteq U$, since the two other two cases are essentially the same.

Suppose not $\cdot \vdash R \trianglelefteq U$.
Then $\cdot \vdash R \trianglelefteq U$ fails, by the Termination Theorem (5.6.1).
So, there is $V \in [\![R]\!]$ with $V \notin [\![U]\!]$, (Soundness of Failure, Theorem 5.6.11).
But, then $[\![R]\!] \not\subseteq [\![U]\!]$. □

Finally, we combine soundness and completeness to obtain an equivalence between $R \leq U$ and $R \trianglelefteq U$.

Theorem 5.6.13 (Correctness of $\trianglelefteq$)

1. $[\![R]\!] \subseteq [\![U]\!]$ *if and only if* $\cdot \vdash R \trianglelefteq U$.
2. $[\![\Psi_1]\!] \subseteq [\![\Psi_2]\!]$ *if and only if* $\cdot \vdash \Psi_1 \trianglelefteq \Psi_2$.
3. $([\![R_1]\!] \backslash [\![U_1]\!]) \times ([\![R_2]\!] \backslash [\![U_2]\!]) \subseteq [\![U]\!]$ *if and only if* $\cdot \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$.

Proof: From left to right: by the Completeness of $\trianglelefteq$ (above, 5.6.12).
From right to left: by the first part of the Soundness of $\trianglelefteq$ Theorem (5.6.10). □

5.7 Extending inclusion to functions

The difficulty of extending to functions

We have omitted functions from datatypes and datasorts up to now because there is no obvious way to extend the inductive semantics of datasorts to functions. Two naïve attempts follow.

$$\begin{aligned} [\![R \to S]\!] &= \{V{:}A \to B \mid \forall V_1 \in [\![R]\!]. V\ V_1 \mapsto^* V_2 \text{ implies } V_2 \in [\![S]\!]\} \\ [\![R \to S]\!] &= \{\lambda x.M \mid x{\in}R \vdash M \in S\} \end{aligned}$$

The first attempt fails because the quantification $\forall V_1 \in [\![R]\!]$ results in the semantics no longer being inductive on terms: V_1 is not a subterm of V. The semantics cannot be inductive on the structure of types due to the recursive nature of datasorts. The second attempt fails because

it depends on the sorting judgment, which in turn depends on the subsorting judgment, which ultimately depends on the semantics of datasorts.

The following example illustrates more concretely the strange situations that can arise if we include functions in systems of recursive types without carefully considering the consequences. Suppose we have a language with pure total functions. Then, we should expect that the function space $R \to S$ is empty exactly when R is non-empty and S is empty, since then there are elements in R, but no elements in S to map them to. Applying this to following declarations yields $r \to r_{\mathsf{emp}}$ is empty iff r is non-empty.

$$\begin{array}{rcl} a & = & c\,(a \to a) \\ r_{\mathsf{emp}} & = & \cdot \\ r & = & c\,(r \to r_{\mathsf{emp}}) \end{array}$$

But, $r = c\,(r \to r_{\mathsf{emp}})$ so r is empty iff r is non-empty! Thus, there seems to be no sensible way to define whether $r \leq r_{\mathsf{emp}}$ holds: it is equivalent to its own negation. One view of this situation is that the contravariance associated with functions leads to the possibility that there are no sets which satisfy the declared equations in a particular signature. (Without such contravariance, monotonicity ensures sets can always be constructed which satisfy a signature, via a fixed point construction.)

Contravariance, negative information, and partial functions

However, real programs do make use of datatypes containing function types, and it seems a major restriction not to allow refinements of such datatypes. Reference types also occur frequently in datatypes, and result in similar complications: we can consider a type like A **ref** to be equivalent to $(1 \to A) \times (A \to 1)$. In the previous work on refinement types by Freeman [Fre94] function sorts are allowed in datasort declarations only when the argument sorts do not involve any of the datasorts currently being defined. This avoids the possibility of datasort declarations making use of recursion in contravariant positions, while still allowing refinements to be declared for some datatypes involving functions and references. However, in practice this limitation seems likely to be restrictive, and it is not so clear that the restriction is necessary. An analysis of the algorithm used by Freeman indicates that it is at least basically well behaved when this restriction is removed: it would still always terminate, and there do not seem to be inclusions that would be accepted that would have undesirable consequences such as sort preservation failing.

If we analyze the above example, another view is that the problem arises because our criteria for emptiness of the function space $R \to S$ makes use of negative information: that R is *not* empty. If we have a system for determining inclusions that ensures that positive information like $R \leq U$ never depends on negative information like $R' \not\leq U'$, then we can be sure that no such contradiction arises (emptiness of R corresponds to the case where $U = \cdot$). Of course, this would leave us with no criteria for determining that a function sort is empty. In the case of pure functions, this seems somewhat unsatisfactory since there are clearly some pure function spaces that are not inhabited. However, if we commit ourselves to only considering partial function spaces, then this is not an issue: every function space is inhabited, namely by a function that never returns a result. Thus, the issue highlighted by the example above does not arise for partial function spaces. Since our main interest is in the design of sorts for ML,

we are mostly only interested in partial (and effectful) function spaces anyway. Thus, we avoid the issue highlighted above by focusing on partial functions.

Previous work on recursive types with partial functions

This decision by itself does not immediately solve all the issues related to datasort declarations which are recursive in contravariant positions. However, the combination of recursive types and partial function types is well understood, in particular in the field of denotational semantics.

For example, MacQueen, Plotkin and Sethi [MPS86] interpret types that include partial functions as *ideals* which are certain sets of elements in an appropriate domain, and show that recursive types can be interpreted as fixed points in this domain. Aiken and Wimmers [AW93] build on this ideal model of types, and explicitly consider the subtyping relationship generated by it for types which are specified via equations which roughly correspond to our datasort declarations. They include a subtyping algorithm but require some restrictions on occurrences of intersections and unions that are not appropriate in our case.

Other notable work on recursive types with partial functions includes that of Amadio and Cardelli [AC93], who use a model based on partial equivalence relations. They also present a subtyping algorithm that uses assumptions for goals in progress to prevent non-termination (our inclusion algorithm is partly based on theirs), although their types do not include any form of union, which would be required in order to represent types corresponding to our datasorts.

The work of Frisch, Castagna and Benzaken [FCB02] similarly semantically constructs an appropriate notion of subtyping in the presence of recursive types, partial non-deterministic functions, intersections, unions and constructors. They use finite functions as an approximation to the actual function space of their language, and demonstrate that the resulting subtyping relationship is appropriate. Alas, we cannot directly use their results, since their subtyping relation includes distributivity and hence is inappropriate when functions may involve effects.

There is a great deal of other work that is relevant to the combination of recursive types and partial functions, and those mentioned above are just a few that seem most relevant to the current work. While it appears that none of these works include a refinement restriction, adding this restriction should only simplifies the situation (although without the restriction the issues related to pure functions do not seem so relevant since non-termination naturally arises via an embedding of the untyped λ-calculus). Thus, it seems that it should be possible to define a sensible subsorting relation in the presence of recursive datasort declarations that include partial functions.

Our design

However, we have chosen not to base our subtyping relationship directly on a denotational semantics like those mentioned above. This is for three main reasons. First, to obtain an appropriate subsorting relationship such a semantics would need to include a suitably rich form of effects, such as dynamically allocated reference cells, and is thus likely to be quite involved. Secondly, in what follows we prefer to give an operational semantics for our language, and having two semantics would be awkward unless they are shown to be equivalent, which is also likely to be involved. Thirdly, such a semantics would almost certainly validate the following distributivity rule, since it is sound in the presence of effects.

$$(R_1 \to S) \,\&\, (R_2 \to S) \;\leq\; (R_1 \vee R_2) \to S$$

While this rule is sound, it destroys the orthogonality of the inclusion rules. Worse, it leads to some serious performance issues due to the need to perform "splitting" of the sorts of function arguments into unions of minimal components during sort checking, as was done by Freeman [Fre94] (splitting will be considered in detail in Section 6.1).

We have thus chosen to simply combine the subsorting relationship in this chapter with the subsorting relationship for functions used in previous chapters. The rules for functions are just the standard contravariant/covariant rule, along with algorithmic rules for breaking down intersections (closely following the rules in Section 3.7) and corresponding rules for breaking down unions of sorts for goals of the form $\Theta \vdash R \trianglelefteq U \sqcup S$. While this is a little ad-hoc, it appears to result in a subsorting relationship that is practical and predictable, based on experience so far with the implementation. However, further experience is needed to be sure that there are no situations where important inclusions are rejected by our formulation.

Interestingly, this subsorting relation corresponds exactly with that obtained with an extension of the semantics of Frisch, Castagna and Benzaken, and in the next section we will take advantage of this correspondence to prove basic properties of subsorting. It would be possible to instead define subsorting in terms of this semantics, but we have chosen not to since some features of this semantics have been explicitly chosen to make this correspondence precise, and are difficult justify otherwise.

Formally, we extend the previous definitions of types and sorts as follows.

$$\begin{array}{rrcl} \text{Types} & A, B & ::= & \ldots \mid A \to B \\ \text{Sorts} & R, S & ::= & \ldots \mid R \to S \end{array}$$

Valid types

$$\frac{\vdash_\Sigma A : \mathsf{type} \qquad \vdash_\Sigma B : \mathsf{type}}{\vdash_\Sigma A \to B : \mathsf{type}}\ \mathsf{typarrow}$$

Valid refinements

$$\frac{\vdash_\Sigma R \sqsubset A \qquad \vdash_\Sigma S \sqsubset B}{\vdash_\Sigma R \to S \sqsubset A \to B}\ \mathsf{srtarrow}$$

The subsorting rules for functions follow. Here, we write $\Theta \vdash R \trianglelefteq S$ as an abbreviation for the form $\Theta \vdash R \trianglelefteq \cdot \sqcup S$.

$$\frac{\Theta \vdash S_1 \trianglelefteq R_1 \qquad \Theta \vdash R_2 \trianglelefteq S_2}{\Theta \vdash R_1 \to R_2 \trianglelefteq S_1 \to S_2} \qquad \frac{\Theta \vdash R_1 \trianglelefteq S_1 \to S_2}{\Theta \vdash R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2} \qquad \frac{\Theta \vdash R_2 \trianglelefteq S_1 \to S_2}{\Theta \vdash R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}$$

$$\frac{R \sqsubset A_1 \to A_2 \qquad \Theta \vdash R \trianglelefteq S_1 \qquad \Theta \vdash R \trianglelefteq S_2}{\Theta \vdash R \trianglelefteq S_1 \,\&\, S_2} \qquad \frac{R \sqsubset A_1 \to A_2}{\Theta \vdash R \trianglelefteq \top^{A_1 \to A_2}}$$

$$\frac{R \sqsubset A_1 \to A_2 \qquad \Theta \vdash R \trianglelefteq U}{\Theta \vdash R \trianglelefteq U \sqcup S} \qquad \frac{R \sqsubset A_1 \to A_2 \quad \Theta \vdash R \trianglelefteq S}{\Theta \vdash R \trianglelefteq U \sqcup S}\,(U \neq \cdot)$$

With the extension to function sorts this judgment can serve as the main subsorting judgment for our language: it can compare any two sorts. This is contrast to previous work on refinements for ML which only allowed restricted occurrences of functions in the judgment that compared recursive definitions, and hence required a separate judgment for the main subsorting judgment of the language, with some duplication between the two judgments.

The implementation takes advantage of this by using only a single subsorting function. This function is actually used in two ways: our implementation first compares datasort declarations to determine a lattice of equivalence classes of refinements for a datatype, and then uses this lattice to determine subsorting instances during sort checking of expressions. However, it avoids duplicating code between these two forms of subsorting by using a single generalized function which includes a parameter that is a function for comparing refinements of base types. See Section 8.3 for more details.

The termination of the extended algorithm requires only a small extension to to the previous termination proof.

Theorem 5.7.1 (Termination of $\trianglelefteq$ with Functions)
There is no infinite sequence $J_1, J_2, \ldots$ with each J_n a conclusion of an instance of one of the algorithmic inclusion rules, and J_{n+1} one of the corresponding premises.

Proof: We modify the well order in the previous termination proof (Theorem 5.6.1) as follows.

We add two new well orders to the end of the lexicographic ordering for syntactic inclusion of R and U respectively in the form $\Theta \vdash R \trianglelefteq U$. □

As discussed above, there is no straightforward way to extend the previous definition of subsorting to the functions of our language: once we have function values like $\lambda x.M$ the semantics can no longer be inductive on values. Thus, for sorts which involve functions we instead define $R \leq S$ to hold exactly when $\cdot \vdash_\Sigma R \trianglelefteq S$. This is somewhat unsatisfactory, and in the next section we consider an inductive semantics that validates the same inclusions as $\cdot \vdash_\Sigma R \trianglelefteq S$, but which uses quite different constructs in place of the actual function values. We could instead define $R \leq S$ in terms of that inductive semantics, but it seems preferable to use the algorithmic judgment $\cdot \vdash_\Sigma R \trianglelefteq S$ because the function values in the inductive semantics have been explicitly designed to match the algorithmic judgment, and so conceptually the algorithmic judgment comes first.

5.8 An inductive counterexample-semantics with functions

Defining $R \leq S$ in terms of the algorithmic judgment $R \trianglelefteq S$ is somewhat unsatisfactory, and fact earlier we criticized previous work on refinements because it required the programmer to understand the inclusion algorithm. Some form of declarative specification would help the programmer to reason about which inclusions hold. The inductive semantics serves this purpose well in the absence of functions, and also allows relatively straightforward proofs of basic properties of the subsorting relationship. In this section we consider an extension of the inductive semantics to function sorts that serves both these purposes, but which is non-standard in the sense that it replaces the actual function values by artificial forms. These artificial "function values" are explicitly designed to produce the same inclusions as the subsorting algorithm, and to allow the semantics to be inductive on the structure of values.

Counterexamples for inclusions

If we consider the way the inductive semantics of Section 5.4 is likely to be used to reason about inclusions in the absence of function sorts, an important concept is that of a counterexample: an inclusion $R \leq S$ holds whenever there is no value V that is in $[\![R]\!]$ but not in $[\![S]\!]$. Such reasoning is very intuitive and is also nicely compositional: the value V may be used to construct larger counterexamples as follows.

- If cR in $\mathsf{body}(\rho)$ and cS in $\mathsf{body}(u)$, and this is the only occurrence of cS in $\mathsf{body}(u)$, then cV is a counterexample for $\rho \leq u$.

- If R' is non-empty and contains V', then (V,V') is a counterexample for $R \times R' \leq S \times S'$ for any S'.

The situation is somewhat more complicated for inclusions involving unions, particularly for products, but the basic idea remains the same: counterexamples are built from smaller counterexamples for "sub-problems".

However, constructing counterexamples for inclusions like $R_1 \to R_2 \leq S_1 \to S_2$ seems more difficult: for example, if $S_1 \not\leq R_1$ then our algorithm will reject the inclusion $R_1 \to R_2 \leq S_1 \to S_2$, but if we have a a value V that is in $[\![S_1]\!]$ but not in $[\![R_1]\!]$ then there does not seem to be a direct way to construct a function value of the form $\lambda x.M$ that is in $[\![R_1 \to R_2]\!]$ but not $[\![S_1 \to S_2]\!]$. Intuitively, it seems enough to specify that the counterexample "maps V to something invalid" and is otherwise well-behaved. This motivates us to consider whether we can construct a non-standard semantics for sorts that replaces the function values $\lambda x.M$ with artificial "function values" that are designed to be particularly appropriate for characterizing the essential features of counterexamples. For example, a counterexample for the inclusion $R_1 \to R_2 \leq S_1 \to S_2$ could be simply written as $V \mapsto !$, indicating "a function value that maps V to something invalid" (and is otherwise well-behaved). Such a non-standard semantics would extend the intuitive compositional reasoning of the earlier inductive semantics to datasort declarations involving function sorts. However, it would do so at the cost of requiring the use of a different semantics from the standard semantics of the language when reasoning about inclusions.

Proving basic properties of subsorting

The lack of an inductive semantics is also unsatisfactory because we can no longer use it to show basic properties of the subsorting relationship $R \leq S$ such as reflexivity, transitivity, and that $R_1 \& R_2$ is a greatest lower bound. These properties are essential for the proofs in Chapter 6. Alas, proving these properties seems difficult without an inductive semantics: the techniques used in Chapter 3 do not extend easily, particularly to the judgment for products. Formulating the induction hypotheses in terms of failures of the inclusion algorithm (as in the proof of completeness, Theorem 5.6.12) seems to allow proofs of reflexivity and transitivity, but the property that $R_1 \& R_2$ is greater than any lower bound seems resistant to proof even with this technique. To use an inductive semantics for such proofs only requires that the semantics and the algorithm agree on which inclusions hold: it does not matter whether the semantics of functions coincides with the actual functions included in the language.

Designing a non-standard inductive semantics based on counterexamples

We are thus motivated to consider the design of a non-standard semantics that validates the same inclusions as our algorithm, and is inductive on the structure of values, with non-standard forms in place of the standard function values $\lambda x.M$. These non-standard "function values" will be "finite characterizations" of functions that specify the potential behavior of the function for some finite number of input and output values. This is essentially the approach to defining subtyping inductively proposed by Frisch, Castagna and Benzaken [FCB02], although in their case the semantics is still constructed mostly "from first principles", while in our case our semantics is explicitly designed to allow counterexamples to be formed whenever an inclusion is rejected by our algorithm. Interestingly, our semantics turns out to be roughly a generalization of that of Frisch, Castagna and Benzaken with constructs that have an appealing duality. The main purpose of this semantics in the context of this dissertation is to allow proofs of properties of the algorithmic subsorting judgment. However, it is also intended to help programmers to determine which subsorting relationships should hold, since it allows counterexamples to be constructed in a relatively intuitive way.

The extension of the inductive semantics to functions is designed so that whenever an inclusion $R \leq U$ does not hold, in the semantics there is a "characterization" of a counterexample which is contained in R but not U. For example, $\texttt{pos} \to \texttt{pos} \leq \texttt{nat} \to \texttt{pos}$ does not hold, and a characterization of a counterexample is: "a function which maps `z` to `z`". Here the second `z` can be replaced by any invalid value, so is not essential to the counterexample, and we could instead write it as: "a function which maps `z` to something invalid". These characterizations explicitly mention all values involved in the counterexample, hence the semantics remains inductive. The key observation here is that whenever an inclusion does not hold, a counterexample can always be characterized in terms of some finite set of values inhabiting the argument sorts and result sorts.

This focus on inductive counterexamples appears to be both a powerful and intuitive technique, and is closely related to the work of Frisch, Castagna and Benzaken [FCB02]. However, their focus is slightly different, mostly because they start from an inductive semantics, and later demonstrate that their full semantics validates the same inclusions, while we focus on the relationship to the algorithmic subsorting judgment, which naturally leads to a greater focus on counterexamples (similar to the proof of completeness without functions, Theorem 5.6.12).

5.8.1 Syntax and semantics

In the inductive semantics of Frisch, Castagna and Benzaken [FCB02], the values inhabiting a function space are finite sets of pairs $M \mapsto N$ of values, with each pair specifying that the function potentially maps the first value to the second. This is the starting point for the semantics below.

We differ in that we include only one of M and N from each pair, since (in our case) only one component is "essential" in any counterexample for $R_1 \to R_2 \not\leq S_1 \to S_2$. Thus, we include constructs: $M \mapsto !$ and $? \mapsto N$. The first means roughly "M maps to something invalid (outside every sort)", while the second means "something valid (in every sort) maps to N". We also include a construct $M \wedge N$ for combining two counterexamples, producing a counterexample that will non-deterministically choose to act like M or like N each time it is used, and hence is only contained in those sorts that contain both M and N. We also have the zero-ary form $\top$ which indicates a function that never returns a result (i.e. that non-terminates for every input), and is contained in every function sort $R \to S$. It is required as a counterexample when we have an empty union on the right, i.e., $R \to S \leq \cdot$. These constructs allow us to form lists of characteristics for a counterexample, similar to the sets of pairs in the inductive semantics of by Frisch, Castagna and Benzaken.

Due to the omission of distributivity rules, we have fewer inclusions than Frisch, Castagna and Benzaken, and thus we need more values to serve as counterexamples. Interestingly, adding dual constructs $M \vee N$ and $\bot$ to $M \wedge N$ and $\top$ results in exactly the required counterexamples for our algorithmic subsorting rules. Thus, it seems that our subsorting relationship is less ad-hoc than it might have appeared at first. The construct $M \vee N$ roughly corresponds to a form of "don't know" non-determinism: either M or N is chosen depending on what is required by the context, hence $M \vee N$ is contained in every sorts that contains either M or N. This may be thought of as a form of effect, although it is certainly different from the actual effects usually included in an ML-like language. This difference does not matter here: what is important is that $\vee$ allows counterexamples to be constructed for inclusions that are rejected, such as those which require distributivity. For example, the distributivity instance

$$(R \to S_1) \,\&\, (R \to S_2) \leq R \to (S_1 \,\&\, S_2)$$

is rejected unless either

$$S_1 \leq S_2 \quad \text{or} \quad S_2 \leq S_1.$$

If V_1 and V_2 are counterexamples for each of these, then $(? \mapsto V_1) \vee (? \mapsto V_2)$ is a counterexample for the distributivity instance.

We also include the zero-ary form $\bot$ of $M \vee N$, which is not contained in any function sort $R \to S$, since we will need it as the counterexample for $\top^{A \to B} \leq S_1 \to S_2$.

We thus extend the syntax of terms (i.e. values), the validity judgment for terms, and the inductive semantics to functions as follows.

Syntax

$$\text{Terms} \quad M, N, V \quad ::= \quad \ldots \mid M \mapsto ! \mid ? \mapsto N \mid M \wedge N \mid \top \mid M \vee N \mid \bot$$

Valid terms (values)

$\vdash_\Sigma M : A$ $\quad M$ is a valid term of type A (where A is a valid type)

$$\frac{\vdash M : A}{\vdash M \mapsto ! : A \to B} \qquad \frac{\vdash N : B}{\vdash ? \mapsto N : A \to B} \qquad \frac{}{\vdash \top : A \to B} \qquad \frac{}{\vdash \bot : A \to B}$$

$$\frac{\vdash M : A \to B \qquad \vdash N : A \to B}{\vdash M \wedge N : A \to B} \qquad \frac{\vdash M : A \to B \qquad \vdash N : A \to B}{\vdash M \vee N : A \to B}$$

Semantics

$$\begin{aligned}
M \mapsto ! \in [\![R \to S]\!] \quad & \text{if } M \notin R \\
? \mapsto N \in [\![R \to S]\!] \quad & \text{if } N \in S \\
M \wedge N \in [\![R \to S]\!] \quad & \text{if } M \in [\![R \to S]\!] \text{ and } N \in [\![R \to S]\!] \\
\top \in [\![R \to S]\!] \quad & \text{(i.e. } \top \text{ in every } R \to S) \\
M \vee N \in [\![R \to S]\!] \quad & \text{if } M \in [\![R \to S]\!] \text{ or } N \in [\![R \to S]\!] \\
\bot \notin [\![R \to S]\!] \quad & \text{(i.e. } \bot \text{ never in } R \to S)
\end{aligned}$$

5.8.2 Soundness and Completeness

We now extend our soundness and completeness proofs to the algorithmic rules for functions, using this inductive "counterexample" semantics for functions. Our main motivation for extending these proofs is that it allows us to easily show basic properties of our algorithmic subsorting relation, such as reflexivity, transitivity, and that $R \& S$ is a greatest lower bound.

First, we require a small lemma relating the semantics of a sort to the type refined by the sort.

Lemma 5.8.1 *If $V \in [\![R]\!]$ and $R \sqsubset A$ then $\vdash V : A$.*

Proof: By induction on R, V lexicographically. □

The previous proof of soundness extends without difficulty to functions with our semantics, although requires a subcase for each form of function value in the case for the covariant-contravariant rule. We call the extension a "soundness" theorem, since it extends the previous soundness theorem (5.6.10) (and similarly for the other extensions of theorems which follow). However, here it plays a slightly different role, since $\leq$ is defined in terms of $\trianglelefteq$. While it still demonstrates that $\trianglelefteq$ is sound with respect to $\subseteq$ in the semantics, here $\trianglelefteq$ conceptually comes before the semantics.

Theorem 5.8.2 (Soundness of $\trianglelefteq$ with Functions)

1. *If $\cdot \vdash_\Sigma R \trianglelefteq U$ and $V \in [\![R]\!]$ then $V \in [\![U]\!]$.*

2. *If* $\cdot \vdash_\Sigma \Psi_1 \trianglelefteq \Psi_2$ *and* $cV \in [\![\Psi_1]\!]$ *then* $cV \in [\![\Psi_2]\!]$.

3. *If* $\cdot \vdash_\Sigma (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$
and $V_1 \in ([\![R_1]\!] \backslash [\![U_1]\!])$ *and* $V_2 \in ([\![R_2]\!] \backslash [\![U_2]\!])$
and $V = (V_1, V_2)$ *then* $V \in [\![U]\!]$.

Proof: By induction on the structure of V and the inclusion derivation, lexicographically. The cases not involving functions are exactly as before (in the proof of Theorem 5.6.10). The proofs of the lemmas used in the previous proof do not require any modification: each is specific to refinements of a particular form of type (a, $\mathbf{1}$ or $A \times B$) and does not require cases to be considered for refinements of other types.

The cases involving functions are as follows.

Case: $\dfrac{\Theta \vdash S_1 \trianglelefteq R_1 \qquad \Theta \vdash R_2 \trianglelefteq S_2}{\Theta \vdash R_1 \to R_2 \trianglelefteq S_1 \to S_2}$

Subcase: $V_1 \mapsto\, ! \in [\![R_1 \to R_2]\!]$

$V_1 \notin [\![R_1]\!]$	Def. $[\![\cdot]\!]$
Suppose $V_1 \in [\![S_1]\!]$	
Then, $V_1 \in [\![R_1]\!]$	Ind. Hyp. (V_1 smaller)
Contradicting $V_1 \notin [\![R_1]\!]$	
So, $V_1 \notin [\![S_1]\!]$	By Contradiction
$V_1 \mapsto\, ! \in [\![S_1 \to S_2]\!]$	Def. $[\![\cdot]\!]$

Subcase: $? \mapsto V_2 \in [\![R_1 \to R_2]\!]$

$V_2 \in [\![R_2]\!]$	Def. $[\![\cdot]\!]$
$V_2 \in [\![S_2]\!]$	Ind. Hyp. (V_2 smaller)
$? \mapsto V_2 \in [\![S_1 \to S_2]\!]$	Def. $[\![\cdot]\!]$

Subcase: $V_1 \wedge V_2 \in [\![R_1 \to R_2]\!]$

$V_1 \in [\![R_1 \to R_2]\!]$	Def. $[\![\cdot]\!]$
$V_2 \in [\![R_1 \to R_2]\!]$	Def. $[\![\cdot]\!]$
$V_1 \in [\![S_1 \to S_2]\!]$	Ind. Hyp. (V_1 smaller)
$V_2 \in [\![S_1 \to S_2]\!]$	Ind. Hyp. (V_2 smaller)
$V_1 \wedge V_2 \in [\![S_1 \to S_2]\!]$	Def. $[\![\cdot]\!]$

Subcase: $\top \in [\![R_1 \to R_2]\!]$

$\top \in [\![S_1 \to S_2]\!]$	Def. $[\![\cdot]\!]$

Subcase: $V_1 \vee V_2 \in [\![R_1 \to R_2]\!]$

$V_1 \in [\![R_1 \to R_2]\!]$ or $V_2 \in [\![R_1 \to R_2]\!]$	Def. $[\![\cdot]\!]$
$V_1 \in [\![S_1 \to S_2]\!]$ or $V_2 \in [\![S_1 \to S_2]\!]$	Ind. Hyp. (V_1, V_2 smaller)
$V_1 \vee V_2 \in [\![S_1 \to S_2]\!]$	Def. $[\![\cdot]\!]$

Subcase: $\bot \in [\![R_1 \to R_2]\!]$

Cannot occur	Def. $[\![\cdot]\!]$

Case: $\dfrac{\Theta \vdash R_1 \trianglelefteq S_1 \to S_2}{\Theta \vdash R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}$

$V \in [\![R_1 \,\&\, R_2]\!]$	Assumed
$V \in [\![R_1]\!]$	Def. $[\![\cdot]\!]$
$V \in [\![S_1 \to S_2]\!]$	Ind. Hyp. (same V, subderivation)

Case: $\dfrac{\Theta \vdash R_2 \trianglelefteq S_1 \to S_2}{\Theta \vdash R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}$

Dual to the previous case.

Case: $\dfrac{R \sqsubset A_1 \to A_2 \qquad \Theta \vdash R \trianglelefteq S_1 \qquad \Theta \vdash R \trianglelefteq S_2}{\Theta \vdash R \trianglelefteq S_1 \,\&\, S_2}$

$V \in [\![R]\!]$	Assumed
$V \in [\![S_1]\!]$	Ind. Hyp. (same V, subderivation)
$V \in [\![S_2]\!]$	Ind. Hyp. (same V, subderivation)
$V \in [\![S_1 \,\&\, S_2]\!]$	Def. $[\![\cdot]\!]$

Case: $\dfrac{R \sqsubset A_1 \to A_2}{\Theta \vdash R \trianglelefteq \top^{A_1 \to A_2}}$

$V \in [\![R]\!]$	Assumed
$R \sqsubset A_1 \to A_2$	Assumed
$\vdash_\Sigma V : A_1 \to A_2$	Lemma 5.8.1
$V \in [\![\top^{A_1 \to A_2}]\!]$	Def. $[\![\cdot]\!]$

Case: $\dfrac{R \sqsubset A_1 \to A_2 \qquad \Theta \vdash R \trianglelefteq U}{\Theta \vdash R \trianglelefteq U \sqcup S}$

$V \in [\![R]\!]$	Assumed
$V \in [\![U]\!]$	Ind. Hyp. (same V, subderivation)
$V \in [\![U \sqcup S]\!]$	Def. $[\![\cdot]\!]$

Case: $\dfrac{R \sqsubset A_1 \to A_2 \qquad \Theta \vdash R \trianglelefteq S}{\Theta \vdash R \trianglelefteq U \sqcup S}\,(U \neq \cdot)$

$V \in [\![R]\!]$	Assumed
$V \in [\![S]\!]$	Ind. Hyp. (same V, subderivation)
$V \in [\![U \sqcup S]\!]$	Def. $[\![\cdot]\!]$

□

For the extension of the completeness proof, we require the following lemmas.

Lemma 5.8.3
If $R, S \sqsubset A \to B$ and $V_1 \in [\![R]\!]$ and $V_2 \in [\![S]\!]$ then $V_1 \vee V_2 \in [\![R \,\&\, S]\!]$.

Proof: By induction on the structure of R, S.

Case: $V_1 \in [\![R_1 \to R_2]\!]$ and $V_2 \in [\![S_1 \to S_2]\!]$.

$V_1 \vee V_2 \in [\![R_1 \to R_2]\!]$	Def. $[\![\cdot]\!]$
$V_1 \vee V_2 \in [\![S_1 \to S_2]\!]$	Def. $[\![\cdot]\!]$
$V_1 \vee V_2 \in [\![(R_1 \to R_2) \,\&\, (S_1 \to S_2)]\!]$	Def. $[\![\cdot]\!]$

Case: $V_1 \in [\![R_1 \,\&\, R_2]\!]$ and $V_2 \in [\![S]\!]$.

$V_1 \in [\![R_1]\!]$ and $V_1 \in [\![R_2]\!]$	Def. $[\![\cdot]\!]$
$V_1 \vee V_2 \in [\![R_1 \,\&\, S]\!]$	Ind. Hyp.
$V_1 \vee V_2 \in [\![R_2 \,\&\, S]\!]$	Ind. Hyp.
$V_1 \vee V_2 \in [\![(R_1 \,\&\, S) \,\&\, (R_2 \,\&\, S)]\!]$	Def. $[\![\cdot]\!]$
$V_1 \vee V_2 \in [\![(R_1 \,\&\, R_2) \,\&\, S]\!]$	Def. $[\![\cdot]\!]$

Case: $V_1 \in [\![R]\!]$ and $V_2 \in [\![S_1 \to S_2]\!]$.

Dual to the previous case. (Although strictly speaking, this case is only required when $R = R_1 \to R_2$ since otherwise it overlaps with the previous case.)

□

Lemma 5.8.4
If $R \sqsubset A \to B$ then $\top \in [\![R]\!]$.

Proof: By induction on R.

□

Lemma 5.8.5
If $R \sqsubset A \to B$ and $V_1 \in [\![R]\!]$ and $V_2 \in [\![R]\!]$ then $V_1 \wedge V_2 \in [\![R]\!]$.

Proof: By induction on R.

Case: $V_1 \in [\![R_1 \to R_2]\!]$ and $V_2 \in [\![R_1 \to R_2]\!]$.

$V_1 \wedge V_2 \in [\![R_1 \to R_2]\!]$ Def. $[\![\cdot]\!]$

Case: $V_1 \in [\![R_1 \,\&\, R_2]\!]$ and $V_2 \in [\![R_1 \,\&\, R_2]\!]$.

$V_1 \in [\![R_1]\!]$ and $V_1 \in [\![R_2]\!]$ Def. $[\![\cdot]\!]$
$V_2 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$ Def. $[\![\cdot]\!]$
$V_1 \wedge V_2 \in [\![R_1]\!]$ Ind. Hyp.
$V_1 \wedge V_2 \in [\![R_2]\!]$ Ind. Hyp.
$V_1 \wedge V_2 \in [\![R_1 \,\&\, R_2]\!]$ Def. $[\![\cdot]\!]$

Case: $R = [\![\top^{A_1 \to A_2}]\!]$.

$V_1 \wedge V_2 \in [\![\top^{A_1 \to A_2}]\!]$ Def. $[\![\cdot]\!]$

□

Lemma 5.8.6
If $R \sqsubset A \to B$ and $V_1 \notin [\![R]\!]$ or $V_2 \notin [\![R]\!]$ then $V_1 \wedge V_2 \notin [\![R]\!]$.

Proof: By induction on R.

Case: $V_1 \notin [\![R_1 \to R_2]\!]$ or $V_2 \notin [\![R_1 \to R_2]\!]$.

$V_1 \wedge V_2 \in [\![R_1 \to R_2]\!]$ Def. $[\![\cdot]\!]$

Case: $V_1 \notin [\![R_1 \,\&\, R_2]\!]$ or $V_2 \notin [\![R_1 \,\&\, R_2]\!]$.

$V_1 \notin [\![R_1]\!]$ or $V_1 \notin [\![R_2]\!]$ Def. $[\![\cdot]\!]$
$V_1 \wedge V_2 \notin [\![R_1]\!]$ or
$V_1 \wedge V_2 \notin [\![R_2]\!]$ Ind. Hyp.
$V_1 \wedge V_2 \notin [\![R_1 \,\&\, R_2]\!]$ Def. $[\![\cdot]\!]$

Case: $V_2 \notin [\![R_1 \,\&\, R_2]\!]$.

Dual to the previous case.

Case: $V_1 \notin [\![\top^{A_1 \to A_2}]\!]$ or $V_2 \notin [\![\top^{A_1 \to A_2}]\!]$.

Cannot occur Def. $[\![\cdot]\!]$

□

Lemma 5.8.7
If $U, S \sqsubset A \to B$ and $V_1 \notin [\![U]\!]$ or $V_2 \notin [\![U]\!]$ then $V_1 \wedge V_2 \notin [\![U]\!]$.

Proof: By induction on U.

Case: $V_1 \notin [\![\cdot]\!]$ or $V_2 \notin [\![\cdot]\!]$.

Then $[\![\cdot]\!]$ is empty, hence $V_1 \wedge V_2 \notin [\![\cdot]\!]$.

Case: $V_1 \notin [\![U \sqcup S]\!]$.

$V_1 \notin [\![U]\!]$ and $V_1 \notin [\![S]\!]$	Def. $[\![\cdot]\!]$
$V_1 \wedge V_2 \notin [\![U]\!]$	Ind. Hyp.
$V_1 \wedge V_2 \notin [\![S]\!]$	Preceding lemma
$V_1 \wedge V_2 \notin [\![U \sqcup S]\!]$	Def. $[\![\cdot]\!]$

Case: $V_2 \in [\![U \sqcup S]\!]$.

Dual to the previous case.

□

Theorem 5.8.8 (Soundness of Failure of $\trianglelefteq$ with Functions)

1. *If $\Theta \vdash_\Sigma R \trianglelefteq U$ fails then there exists $V \in [\![R]\!]$ such that $V \notin [\![U]\!]$.*
2. *If $\Theta \vdash_\Sigma \Psi_1 \trianglelefteq \Psi_2$ fails*
   then there exists $cV \in [\![\Psi_1]\!]$ such that $cV \notin [\![\Psi_2]\!]$.
3. *If $\Theta \vdash_\Sigma (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$ fails*
   then there exists $V_1 \in [\![R_1]\!]$ and $V_2 \in [\![R_2]\!]$
   such that $V_1 \notin U_1$ and $V_2 \notin U_2$ and $(V_1, V_2) \notin [\![U]\!]$.

Proof: By induction on the termination order of the algorithm. The cases not involving functions are exactly as before (in the proof of Theorem 5.6.12). The cases involving functions are as follows.

Case: $\Theta \vdash R_1 \to R_2 \trianglelefteq S_1 \to S_2$ fails because $\Theta \vdash S_1 \trianglelefteq R_1$ fails.

The induction hypothesis yields $V_1 \in [\![S_1]\!]$ with $V_1 \notin [\![R_1]\!]$.
Then $(V_1 \mapsto !) \in [\![R_1 \to R_2]\!]$ and $(V_1 \mapsto !) \notin [\![S_1 \to S_2]\!]$.

Case: $\Theta \vdash R_1 \to R_2 \trianglelefteq S_1 \to S_2$ fails because $\Theta \vdash R_2 \trianglelefteq S_2$ fails.

The induction hypothesis yields $V_2 \in [\![R_2]\!]$ with $V_2 \notin [\![S_2]\!]$.
Then $(? \mapsto V_2) \in [\![R_1 \to R_2]\!]$ and $(? \mapsto V_2) \notin [\![S_1 \to S_2]\!]$.

Case: $\Theta \vdash R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2$ fails.

Then the algorithm will have checked both

$$\Theta \vdash R_1 \trianglelefteq S_1 \to S_2 \quad \text{and} \quad \Theta \vdash R_2 \trianglelefteq S_1 \to S_2$$

and found that both fail.

Applying the induction hypothesis yields:
$V_1 \in [\![R_1]\!]$ such that $V_1 \notin [\![S_1 \to S_2]\!]$ and also
$V_2 \in [\![R_2]\!]$ such that $V_2 \notin [\![S_1 \to S_2]\!]$.
Then, $V_1 \vee V_2 \in [\![R_1 \,\&\, R_2]\!]$ and $V_1 \vee V_2 \notin [\![S_1 \to S_2]\!]$, as required.

Case: $\Theta \vdash \top^{A_1 \to A_2} \trianglelefteq S_1 \to S_2$ fails because it matches no rule.
Then $\bot \in [\![\top^{A_1 \to A_2}]\!]$ and $\bot \notin [\![S_1 \to S_2]\!]$.

Case: $\Theta \vdash R \trianglelefteq S_1 \,\&\, S_2$ fails because $\Theta \vdash R \trianglelefteq S_1$ fails (with $R \sqsubset A \to B$).
The induction hypothesis yields $V \in [\![R]\!]$ with $V \notin [\![S_1]\!]$.
Thus $V \notin [\![S_1 \,\&\, S_2]\!]$ by the definition of $[\![\cdot]\!]$.

Case: $\Theta \vdash R \trianglelefteq S_1 \,\&\, S_2$ fails because $\Theta \vdash R \trianglelefteq S_2$ fails (with $R \sqsubset A \to B$).
Dual to the previous case.

Case: $\Theta \vdash R \trianglelefteq U \sqcup S$ fails because $\Theta \vdash R \trianglelefteq U$ fails and $\Theta \vdash R \trianglelefteq S$ fails (with $R \sqsubset A \to B$).
The induction hypothesis yields $V_1 \in [\![R]\!]$ with $V_1 \notin [\![U]\!]$ and $V_2 \in [\![R]\!]$ with $V_2 \notin [\![S]\!]$.
Then $V_1 \wedge V_2 \in [\![R]\!]$ and $V_1 \wedge V_2 \notin [\![U \sqcup S]\!]$ (by Lemma 5.8.6 and Lemma 5.8.7).

Case: $\Theta \vdash R \trianglelefteq \cdot$ fails because it matches no rule (with $R \sqsubset A \to B$).
Then $\top \in [\![R]\!]$ (Lemma 5.8.4) and $\top \notin [\![\cdot]\!]$.

□

As before, to prove our completeness result, we combine the soundness of failure theorem with the termination theorem.

Theorem 5.8.9 (Completeness of $\trianglelefteq$ with Functions)

1. *If* $[\![R]\!] \subseteq [\![U]\!]$ *then* $\cdot \vdash R \trianglelefteq U$.
2. *If* $[\![\Psi_1]\!] \subseteq [\![\Psi_2]\!]$ *then* $\cdot \vdash \Psi_1 \trianglelefteq \Psi_2$.
3. *If* $([\![R_1]\!] \backslash [\![U_1]\!]) \times ([\![R_2]\!] \backslash [\![U_2]\!]) \subseteq [\![U]\!]$ *then* $\cdot \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$.

Proof: By contradiction, as before. We assume that the algorithm does not succeed, then use the Termination of $\trianglelefteq$ with Functions Theorem (5.7.1) to show that the algorithm fails, and then use the Soundness of Failure with Functions Theorem (5.8.8) to show that the set inclusion does not hold. □

Finally, we combine soundness and completeness to obtain an equivalence between $[\![R]\!] \subseteq [\![U]\!]$ and $R \trianglelefteq U$, as before.

Theorem 5.8.10 (Correctness of $\trianglelefteq$ with Functions)

1. $[\![R]\!] \subseteq [\![U]\!]$ *if and only if* $\cdot \vdash R \trianglelefteq U$.
2. $[\![\Psi_1]\!] \subseteq [\![\Psi_2]\!]$ *if and only if* $\cdot \vdash \Psi_1 \trianglelefteq \Psi_2$.
3. $([\![R_1]\!] \backslash [\![U_1]\!]) \times ([\![R_2]\!] \backslash [\![U_2]\!]) \subseteq [\![U]\!]$ *if and only if* $\cdot \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U$.

Proof: From left to right: by the Completeness of $\trianglelefteq$ with Functions Theorem (above, 5.8.9). From right to left: by the first part of the Soundness of $\trianglelefteq$ with Functions Theorem (5.8.2). □

Now, we can use this theorem to avoid separately proving many basic properties of $R \trianglelefteq S$, as was required in Section 3.7. Instead, such properties can be inherited from the corresponding properties for sets, as in the following corollary. Separately proving these properties turns out to be surprisingly difficult, and this corollary is our main motivation for introducing the inductive semantics of functions: these properties are required for the proofs in Chapter 6. We include properties for $\Psi_1 \trianglelefteq \Psi_2$ as well, since these will also be required in the proofs.

Corollary 5.8.11 (Properties of $\trianglelefteq$)
The binary relations $\cdot \vdash R \trianglelefteq S$ and $\cdot \vdash \Psi_1 \trianglelefteq \Psi_2$ are reflexive and transitive, and $\cdot \vdash R \trianglelefteq S$ has greatest lower bounds given by $R_1 \,\&\, R_2$ and maximum refinements given by $\top^A$.

Proof:
From the corresponding properties of $\subseteq$, since:

$\cdot \vdash R \trianglelefteq S$ is equivalent to $[\![R]\!] \subseteq [\![S]\!]$
$\cdot \vdash \Psi_1 \trianglelefteq \Psi_2$ is equivalent to $[\![\Psi_1]\!] \subseteq [\![\Psi_2]\!]$

$[\![R \,\&\, S]\!] = [\![R]\!] \cap [\![S]\!]$
$[\![\top^A]\!] = \{M \mid \vdash_\Sigma M : A\}$

□

Chapter 6

Sort checking with datasorts and pattern matching

We now consider sort assignment and sort checking for a language with base sorts defined using datasort declarations. The core of this language is essentially an instantiation of the framework in Chapter 3. The base subsorting relation is now determined by comparing datasort declarations, as described in Chapter 5, rather than being directly declared in a signature. Similarly, the sorts associated with of each constructor c are determined from the datasort declarations. A major difference from Chapter 3 is that we add an elimination construct for datatypes: **case** expressions with sequential pattern matching, which additionally serve as the elimination construct for products, as in ML and Haskell.

Previous work on refinement types for ML [Fre94] only treated the simple single-level case construct, and argued that pattern matching could be compiled into this construct prior to performing sort inference. But this is unsatisfactory because it requires a programmer to reason in terms of this compilation when considering whether their program is sort correct, and worse it requires them to interpret error messages based on this compilation. Thus, it seems desirable to specify sort correctness directly at the source level, including pattern matching. In this chapter we present such a specification, as sort assignment rules for a language with sequential pattern matching. We demonstrate that our language satisfies an appropriate progress and sort preservation theorem, and then present a sort checking algorithm for our language and prove it correct with respect to our sort assignment rules.

But first we treat some issues related to the sorts for **case** expressions and constructors, and in particular our choice to use inversion principles rather than splitting.

6.1 Splitting, inversion principles and sorts of constructors

Splitting

We make a major departure from the previous work on sorts for ML in that we don't perform "splitting" (see Freeman [Fre94]). Splitting breaks all sorts down into unions of minimal components. For example, with the following declarations the sort `bool` would be split into the union of `tt` and `ff`.

```
datasort bool = true | false
datasort tt = true
datasort ff = false
```

We continue this example by considering the sorts for the following function declarations both with and without splitting.

```
(*[ val not <: tt -> ff  &  ff -> tt  ]*)
fun not true = false
  | not false = true

(*[ val g <: bool -> bool  ]*)
fun g y = not (not y)
```

The declaration of `g` is rejected without splitting, because the sort of the parameter `y` is `bool`, and the sort of `not` does not include a component matching this sort. With splitting this declaration is accepted, because we split the sort of the parameter into `tt` and `ff` and check the body of `g` separately for each part. This example could be modified so that it is accepted without splitting by adding the required component to the sort for `not`, i.e. as follows.

```
(*[ val not <: tt -> ff  &  ff -> tt  &  bool -> bool  ]*)
fun not true = false
  | not false = true
```

This is similar to the situation with the removal of the problematic distributivity subsorting rule in Chapter 3, which sometimes results in extra components being required in the sorts of functions. Splitting is related to the following dual of the problematic distributivity rule, which is generally included in systems with both union and intersection types, such as that of Barbanera, Dezani-Ciancaglini and de'Liguoro [BDCdL95], and which appears to be sound in the presence of effects.

$$\frac{}{(R_1 \to S)\,\&\,(R_2 \to S) \leq (R_1 \vee R_2) \to S}$$

As demonstrated by the the above example, splitting leads to more programs being accepted. However, it has a high cost: it requires a case analysis every time a variable is assigned a sort that can be split, and multiple splittable sorts have a multiplicative effect. Further, when a base lattice has many elements the number of components in each split can become quite large: in the worst case exponential in the number of datasort declarations. This means that the efficiency of sort checking in the presence of splitting is likely to be very sensitive to the addition of new datasort declarations.

Inversion principles

We thus choose not to perform splitting. This choice requires us to find an alternative that is easy to specify, predictable and natural for a programmer, and that avoids efficiency problems. The solution presented here appears to meet these goals, but some more experience is required to judge whether programmers find it natural, and whether efficiency is ever a problem.

Our approach is to determine an *inversion principle* for each datasort that is used whenever an object is matched against a constructor. In the simplest case the inversion principle for a datasort is exactly the body of its datasort declaration. E.g. if we have the declaration

$$r = c_1\, R_1 \sqcup c_2\, R_2$$

and we encounter a situation where an object with sort r is matched against a constructor, then we consider the following two cases.

1. The object has the form $c_1\, V_1$ with $V_1 \in R_1$.
2. The object has the form $c_2\, V_2$ with $V_2 \in R_2$.

We only perform the case analysis indicated by an inversion principle when an object is matched against a constructor, unlike splitting which always breaks up sorts into minimal components. Our inversion principles also generally require fewer cases to be analyzed than splitting. For example, consider the following declarations.

```
(*[ datasort nat = bnil | b0 of pos | b1 of nat
         and pos =        b0 of pos | b1 of nat
         and zero = bnil                       ]*)

(*[ inc <:  nat -> pos ]*)
fun inc bnil = b1 bnil
  | inc (b0 x) = b1 x
  | inc (b1 x) = b0 (inc x)
```

With these declarations, the inversion principle for `nat` would have three cases corresponding to the three cases in its definition, i.e., one for each constructor. Thus, the body of each branch of the function `inc` is analyzed only once, with the assignment `x` $\in$ `pos` for the second branch, and the assignment `x` $\in$ `nat` for the third branch. With splitting, the sort `nat` is decomposed into the union of `pos` and `zero`, so the body of the third branch would need to analyzed separately under the assignments `x` $\in$ `pos` and `x` $\in$ `zero`.

The inversion principle for a datasort is determined reasonably directly from the cases in its datasort declaration. This means that a programmer can alter a datasort declaration so that it yields a different inversion principle even though it specifies an equivalent set of values. For example, consider the following variation of the previous datasort declarations.

```
(*[ datasort nat2 = bnil | b0 of pos2 | b1 of zero | b1 of pos2
         and pos2 =        b0 of pos2 | b1 of zero | b1 of pos2
         and zero = bnil                                      ]*)
```

These declarations declare datasorts containing the same values as the declarations above, but the resulting inversion principle for `nat2` would result in the third case for `inc` being analyzed twice: once with `x` $\in$ `pos2` and once with `x` $\in$ `zero`. Thus, it is possible to emulate the case analysis performed by splitting when matching against a constructor by choosing appropriate datasort declarations.

At first it might appear strange to use different inversion principles for datasort declarations that specify the same sets of values. In fact, an earlier version of our implementation always

used the same inversion principle for datasorts containing the same set of values: it always chose the most detailed principle.[1] Our choice to allow different inversion principles seems justified by the control it gives the programmer over the cases that are considered during sort checking. Also, our experience indicates that programmers have a tendency to naturally choose datasort declarations corresponding to the cases that need to be considered in their code. Should the most detailed principle be required, the programmer can always write their datasort declaration in a way that corresponds to this principle. Additionally, using inversion principles that are closely based on the datasort declarations generally makes it easier for a programmer to understand the case analysis that is being performed. This is particularly important when a programmer is faced with an error message generated during sort checking and is determining the source of the error.

Sorts of constructors and multiple datasort declarations

A similar issue arises with the sorts of constructors. Consider the the two previous sets of datasort declarations.

```
(*[ datasort nat  = bnil | b0 of pos  | b1 of nat
         and pos  =        b0 of pos  | b1 of nat
         and zero = bnil                                    ]*)

(*[ datasort nat2 = bnil | b0 of pos2 | b1 of zero | b1 of pos2
         and pos2 =        b0 of pos2 | b1 of zero | b1 of pos2
         and zero = bnil                                    ]*)
```

As before, these declarations define equivalent sets of values. However, the most direct and natural way of assigning sorts to constructors based on these two sets of declarations leads to the following inequivalent sorts for the constructor `b1`. Here, and from now on, `->` is assumed to have higher precedence than `&`.

```
b1 ∈  nat->nat  &  nat->pos
b1 ∈ zero->nat2 & zero->pos2 &
     pos2->nat2 & pos2->pos2
```

The first sort for `b1` is more general than the second: the first is equivalent to `nat->pos` while the second is equivalent to `zero->pos2 & pos2->pos2`, and thus the first is a subsort of the second due to the contravariance of `->`, and the fact that `pos` and `pos2` are equivalent. However, the second is not a subsort of the first. Thus, with the second set of declarations the following example would be rejected.

```
(*[ val prefix1 <: nat2 -> nat2 ]*)
prefix1 x = b1 x
```

[1] The current version of the implementation allows reverting to this behavior via a user-settable flag, although we have yet to find an examples where this is desirable.

We could obtain equivalent results for these two declarations by strengthening the sort of the constructor `b1` for the second declaration. This could be done by applying the following distributivity rule. (This rule is a small variant of the distributivity rule related to splitting near the start of this section, and can easily be derived from that rule.)

$$\frac{}{(R_1 \to S_1) \,\&\, (R_2 \to S_2) \leq (R_1 \vee R_2) \to (S_1 \vee S_2)}$$

An earlier version of the implementation applied this rule to each pair of conjuncts in the constructor sort, but in some cases this proved to have a large performance cost. This is because generally the there is no sort in our language which is equivalent to the union of R_1 and R_2, so instead a least upper bound needs to be constructed by considering all upper bounds of R_1 and R_2, which in general seems to require enumerating all sorts compatible with R_1 and R_2. Such an enumeration has a potentially huge performance cost, and is infeasible for even simple higher-order sorts, and so doesn't seem to be justified.[2] Additionally, forcing the sorts of constructors to be reasonably apparent in the datasort declarations is likely to make it easier for a programmer to follow the sorts associated with their program. Further, should a situation arise where a programmer requires the inversion principle associated with the declaration of `nat2`, but also requires `b1` $\in$ `nat2 -> pos2`, then they can simply include both sets of declarations.

```
(*[ datasort nat2 = bnil | b0 of pos2 | b1 of zero | b1 of pos2
         and pos2 =        b0 of pos2 | b1 of zero | b1 of pos2
         and zero = bnil

    datasort nat  = bnil | b0 of pos  | b1 of nat
         and pos  =        b0 of pos  | b1 of nat                ]*)
```

The second set of declarations actually does not lead to new elements being added to the lattice of refinements, since the declared datasorts are equivalent to elements added by the first declaration. Instead, `nat` becomes a synonym for `nat2` and `pos` becomes a synonym for `pos2`. However, the second declaration does lead to the sort of the constructor `b1` being strengthened to `nat2 -> pos2`.

If we reverse the order of these declarations, so that `nat` and `pos` are defined first, then the definitions of `nat2` and `pos2` result in the inversion principles for `nat` and `pos` being strengthened. Thus, these inversion principles are no longer based on only the components of the original datasort declaration. In general, the inversion principle for a datasort is based on its own declaration, as well as the declarations of all equivalent or larger datasorts. This ensures that the case analysis performed when an object is assigned one sort is at least as accurate as the case analysis that would be performed for a supersort. This "monotonicity of inversion principles" is important because a programmer may have in mind the case analysis corresponding to a particular sort for an object, while the case analysis performed during sort checking could be based on a more precise sort.

Our approach to inversion principles and sorts of constructors has the advantage of allowing the programmer a large amount of control over this aspect of sort checking. More experience is

[2] There is a flag in the implementation that allows reverting to this behavior.

required to determine whether there are situations where this approach is awkward. At worst, our approach may require the programmer to include two datasort declarations for each sort. The control provided by this approach has the additional advantage that it is generally easy to formulate datasort declarations that emulate alternative approaches, so that the impact of such approaches can be judged without needing to implement them directly.

Formal specification of inversion principles and sorts of constructors

The inversion principle for a base refinement ρ is defined formally as the datasort body for the intersection of all base sorts r which are supersorts, as follows.

$$\mathsf{inv}_\Sigma(\rho) \quad = \quad \mathsf{body}_\Sigma(\&\{r \mid \rho \leq r\})$$

Here we use the definitions of $\mathsf{body}_\Sigma(\cdot)$ and $\leq$ from Chapter 5. For convenience, we repeat the definition of $\mathsf{body}_\Sigma(\cdot)$ below.

$$\begin{aligned}
\mathsf{body}(r) &= \Psi && \text{when } r = \Psi \text{ in } \Sigma \\
\mathsf{body}(\rho_1 \& \rho_2) &= \sqcup\{c(R_1 \& R_2) \mid cR_1 \text{ in } \mathsf{body}(\rho_1), cR_2 \text{ in } \mathsf{body}(\rho_2)\} \\
\mathsf{body}(\top^a) &= \sqcup\{c\top^A \mid cA \text{ in } D\} && \text{when } a = D \text{ in } \Sigma
\end{aligned}$$

The definition of $\mathsf{inv}_\Sigma(\rho)$ can result in redundant and empty components in the inversion principle, which can easily be removed as an optimization in an implementation. As mentioned above, this definition includes supersorts to ensure that inversion principles are monotonic (this monotonicity will be formally proved later, in Lemma 6.6.10).

By contrast, the sorts for constructors are always determined directly from the datasort bodies for a base sort, rather than the intersection of all supersorts. Formally, we have the following judgment, which will later be used when assigning sorts to constructor applications (in Section 6.5).

$\vdash_\Sigma cS \rightarrowtail \rho$ Constructor c maps from S to ρ.

$$\frac{cR \text{ in } \mathsf{body}_\Sigma(\rho) \qquad S \trianglelefteq R}{\vdash_\Sigma cS \rightarrowtail \rho}$$

If instead of $\mathsf{body}_\Sigma(\rho)$ we used $\mathsf{inv}_\Sigma(\rho)$, the sorts obtained would be weaker. For example, with the earlier declarations of `nat`, `pos`, `zero`, `nat2` and `pos2`, using $\mathsf{body}_\Sigma(\rho)$ leads to the following holding (removing the redundant sorts arising from `nat` and `nat2`).

```
b1 nat  ⇀ pos
b1 zero ⇀ pos2
b1 pos2 ⇀ pos2
```

Using $\mathsf{inv}_\Sigma(\rho)$ would lead to only the following holding (simplifying intersections of base sorts appropriately).

```
b1 zero ⇀ pos
b1 pos2 ⇀ pos
b1 zero ⇀ pos2
b1 pos2 ⇀ pos2
```

This is because `pos2` is equivalent to `pos`, hence `pos2` is included as one of the supersorts when we calculate $\mathsf{inv}_\Sigma(\texttt{pos})$. This results in strictly weaker sorts for `b1`, and in fact $\texttt{b1}\,\texttt{nat} \rightarrowtail \texttt{pos}$ is strictly stronger than all the other sorts for the constructor.

We remark that our judgment $\vdash_\Sigma c\,S \rightarrowtail \rho$ is not strictly necessary, since in what follows we could make use of $\mathsf{body}_\Sigma(\rho)$ directly instead. Our reason for introducing it is partly as a notational convenience, but it is also to slightly separate this aspect from what follows, to make it easier to consider alternative ways of defining the sorts for constructors. Further, the sort assignment rules which follow have their dependence on the signature Σ isolated to uses of three forms: the function $\mathsf{inv}_\Sigma(\rho)$ and the judgments $\vdash_\Sigma c\,S \rightarrowtail \rho$ and $\vdash_\Sigma R \trianglelefteq U$ (with $U = S$ or $U = \cdot$). It seems likely that there are other sensible ways to define these three forms, thus we anticipate the generalization of what follows to a form that is parameterized with respect to the alternative definitions of these forms. However, it seems clear that such a generalization would require some constraints to ensure that such alternative definitions are sensible and consistent, and formulating these constraints appropriately does not appear to be easy. We plan to consider this further in future work.

6.2 Syntax with pattern matching

We now consider the syntax of our language $ML^{\&\mathbf{case}}$ with datasorts defined in a signature, as in Chapter 5, and with a **case** elimination construct which allows a sequence of patterns. Each pattern may eliminate many constructor applications, as well as products. The sequence of patterns may overlap, in which case the first matching branch is chosen (as in ML or Haskell).

This language is as an extension of the language in Chapter 5, and in particular uses the same signatures and subsorting. The language of terms is significantly extended to include variables, functions, and our **case** construct. We also include recursion via a fixed-point term construct, and corresponding expression variables u, so that appropriate functions over recursive datatypes can be defined. This language is expressive enough to allow terms to be constructed corresponding to many functions from real ML programs (although not those using effects or parametric polymorphism). We do not include **let** here because in the absence of parametric polymorphism it can be considered as syntactic sugar for a **case** with a single branch.

$$\begin{array}{rrcl}
\text{Expression} & M & ::= & x \mid \lambda x{:}A.M \mid M_1\,M_2 \\
& & & \mid c\,M \mid (M_1, M_2) \mid () \mid \mathbf{case}\ M\ \mathbf{of}\ \Omega \\
& & & \mid u \mid \mathbf{fix}\ u.M \\
\text{Branches} & \Omega & ::= & \overset{A}{\cdot} \mid (P \Rightarrow M \mid \Omega) \\
\text{Pattern} & P & ::= & x \mid c\,P \mid (P_1, P_2) \mid ()
\end{array}$$

A case expression $\mathbf{case}\,M\,\mathbf{of}\,\Omega$ consists of a term M that is the subject of the **case**, and a sequence of branches Ω of the following form.

$$P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n \mid \overset{A}{\cdot}$$

Here, the type annotation A on $\overset{A}{\cdot}$ is required to ensure uniqueness of types when there are no branches. We generally omit the A when the list of branches is non-empty, or when A is otherwise clear from the context, or of little interest. As usual, we generally omit the "$\cdot$" for non-empty lists.

6.3 Typing and pattern contexts

Our presentation of the typing for $ML^{\&\mathbf{case}}$ is based on a correspondence between pattern matching and formulations of logics in the sequent-calculus style of Gentzen [Gen35, Gen69]. This is an instance of the Curry-Howard isomorphism [How80] between typed languages and logics. The correspondence between patterns and left rules has been observed previously: see, for example, the work of Breazu-Tannen, Kesner and Puel [KPT96].

We have chosen this style of presentation because we believe it leads to the correct logical view of the role of pattern matching in functional languages. This becomes critical when we present the sort assignment rules for $ML^{\&\mathbf{case}}$: other common approaches to the typing of pattern matching do not seem to extend nearly as elegantly to our sorts. Our focus is on capturing the essence of pattern matching as intuitively and elegantly as possible, and thus we do not directly relate our presentation to any particular sequent calculus.

This style of presentation requires a generalization of contexts to include assumptions which assign a type to a pattern rather than to a variable. The patterns in such assumptions are broken down by rules which correspond to sequent calculus left rules, eventually leading to an ordinary set of assignments of types to variables.

To allow our system to be more easily compared to other approaches to typing of patterns, we require that all pattern assumptions be broken down into ordinary assumptions prior to considering the constructs in the body of a branch. This requirement is not strictly necessary, but without it there are some complications related to ill-typed pattern matches if none of the variables bound by the pattern are used. In the sort assignment rules in Section 6.5 no such complications arise, and we allow more flexible reasoning by a programmer by allowing pattern assumptions to be broken down later as needed.

For the type system we enforce the requirement by including both ordinary contexts and pattern contexts in our language, with the latter only being used when breaking down assumptions prior to checking the body of a branch. The syntax for these two forms of contexts follows. We recall that variables of the form u expression variables, and are only bound by the fixed-point construct, hence only occur in variable contexts and never in patterns.

$$\begin{array}{rrcl} \text{Variable Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, u{:}A \\ \text{Pattern Contexts} & \Phi & ::= & \cdot \mid \Phi, P{:}A \end{array}$$

We require variables to appear at most once in a variable context. We omit the formal validity judgments for contexts. The validity judgment for variable contexts is standard: it requires that the type assigned to each variable be valid.

The validity judgment for pattern contexts similarly only requires that the type assigned to each pattern be valid. This allows odd pattern assumptions such those which are ill-typed, like $(x,y) : 1$, and those which repeat variables, such as $(x,x) : A \times B$. The typing rules ensure that such assumptions can never appear as part of typing derivation: we consider the validity of these assumptions to be dual to the validity of terms, and hence it is appropriate that the validity of both are judged by the typing rules.

Formally we have the following judgments for the validity of terms.

Valid terms

$\Gamma \vdash_\Sigma M : A$	Term M has type A in under Γ, Σ.
$\Gamma \vdash_\Sigma A \bowtie \Omega : B$	Branches Ω map from A to B under Γ, Σ.
$\Gamma; \Phi \vdash_\Sigma M : B$	Term M has type B under Γ, Φ, Σ.

The typing rules for these judgments appear in Figure 6.1.

6.4 Reduction semantics

We now present a reduction style semantics for our language following Wright and Felleisen [WF94], thus in a similar style to Chapter 4. We distinguish the following terms as values.

$$\text{Values} \quad V, W \quad ::= \quad x \mid \lambda x.M \mid () \mid (V,W) \mid c\,V$$

We have the following evaluation contexts within which a reduction may occur.

$$E \quad ::= \quad [\,] \mid E\,M \mid V\,E \mid c\,E \mid (E,M) \mid (V,E) \mid \mathbf{case}\ E\ \mathbf{of}\ \Omega$$

As in Chapter 4 we write $E[M]$ to indicate the term obtained by replacing the hole $[\,]$ in E by M. A program state is either a term M, or a single special state indicating that a match exception has occurred due to a **case** subject not matching the pattern of any branch.

$$\text{Program States} \quad \pi \quad ::= \quad M \mid \mathit{MatchException}$$

We write $M \mapsto \pi$ for a one-step computation, defined by the following reduction rules. Each rule reduces a redex N that appears in an evaluation position in the term M, i.e. $M = E[N]$ for some E. We maintain the invariant that M does not contain free variables x or u.

$$\begin{array}{rcll}
E[(\lambda x.M)\,V] & \mapsto & E[\{V/x\}M] & \\
E[\mathbf{case}\ V\ \mathbf{of}\ P \Rightarrow M \mid \Omega] & \mapsto & E[\sigma M] & (\text{if } V = \sigma P) \\
E[\mathbf{case}\ V\ \mathbf{of}\ P \Rightarrow M \mid \Omega] & \mapsto & E[\mathbf{case}\ V\ \mathbf{of}\ \Omega] & (\nexists \sigma.V = \sigma P) \\
E[\mathbf{case}\ V\ \mathbf{of}\ \cdot] & \mapsto & \mathit{MatchException} & \\
E[\mathbf{fix}\ u.M] & \mapsto & E[\{\mathbf{fix}\ u.M/u\}M] &
\end{array}$$

The rules for matching against a pattern check whether the case subject V is a substitution instance of the pattern P, and if so applies the corresponding substitution $\sigma = \{V_1/x_1, \ldots, V_n/x_n\}$ to the body of the branch. We feel that this elegantly captures the semantics of pattern matching. We remark that every pattern P is also a syntactically a term M. We consider that patterns are distinguished from terms by the position that they appear in a term rather than intrinsically by their form. This is similar to the standard treatment of variables: variable occurrences are either terms or binding occurrences depending on where they appear in a term.

We now consider progress for well-typed terms and preservation of types. We only sketch a proof of this theorem, since our main interest is in the corresponding result for sorts. We first require substitution lemmas for values, expressions and patterns.

$$\frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash x : A} \qquad \frac{(a = D) \text{ in } \Sigma \quad c\,B \text{ in } D \quad \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma c\,M : a}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

$$\frac{u{:}A \text{ in } \Gamma}{\Gamma \vdash u{:}A} \qquad \frac{\Gamma, u{:}A \vdash M : A}{\Gamma \vdash \mathbf{fix}\ u.M : A}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M,N) : A \times B} \qquad \frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \bowtie \Omega : B}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega : B}$$

. . . . . .

$$\frac{\Gamma; P{:}A \vdash M : B \qquad \Gamma \vdash A \bowtie \Omega : B}{\Gamma \vdash A \bowtie (P \Rightarrow M \mid \Omega) : B} \qquad \frac{}{\Gamma \vdash A \bowtie (\overset{B}{\bullet}) : B}$$

. . . . . .

$$\frac{\Gamma; (\Phi, P_1{:}A_1, P_2{:}A_2) \vdash M : B}{\Gamma; (\Phi, (P_1, P_2) : A_1 \times A_2) \vdash M : B} \qquad \frac{\Gamma; \Phi \vdash M : B}{\Gamma; (\Phi, ():1) \vdash M : B}$$

$$\frac{a = D \text{ in } \Sigma \qquad c\,A \text{ in } D \qquad \Gamma; (\Phi, P{:}A) \vdash_\Sigma M : B}{\Gamma; (\Phi, c\,P{:}a) \vdash_\Sigma M : B}$$

$$\frac{\Gamma \vdash M : B}{\Gamma; \cdot \vdash M : B} \qquad \frac{(\Gamma, x{:}A); \Phi \vdash M : B}{\Gamma; (\Phi, x{:}A) \vdash M : B}$$

Figure 6.1: Typing rules for terms, branches and patterns.

Lemma 6.4.1 (Value substitution for types)
If $V{:}A$ *and* $(\Gamma, x{:}A); \Phi \vdash M : B$
then $\Gamma; \Phi \vdash \{V/x\}M \in B$.

Proof: (sketch)
By induction on the structure of the derivation of $(\Gamma, x{:}A); \Phi \vdash M : B$.

The case for a value variable matching x simply uses the derivation of $V{:}A$ (via weakening). The remaining cases simply rebuild the derivation. □

Lemma 6.4.2 (Expression substitution for types)
If $M{:}A$ *and* $(\Gamma, u{:}A); \Phi \vdash N : B$
then $\Gamma; \Phi \vdash \{M/u\}N \in B$.

Proof: (sketch)
By induction on the structure of the derivation of $(\Gamma, u{:}A); \Phi \vdash N : B$.

The case for an expression variable matching u simply uses the derivation of $M{:}A$ (via weakening). The remaining cases simply rebuild the derivation. □

Lemma 6.4.3 (Pattern substitution for types)
If $V{:}A$ *and* $V = \sigma P$ *and* $\bullet; P{:}A \vdash M : B$
then $\sigma M \in B$.

Proof: (sketch)
By induction on the structure of the derivation of $\bullet; P{:}A \vdash M : B$, generalizing to the following.

If $V_1{:}A_1$ and ... and $V_n{:}A_n$, and $V_1 = \sigma P_1$ and ... and $V_n = \sigma P_n$,
and $\bullet; (P_1{:}A_1, \ldots, P_n{:}A_n) \vdash M : B$
then $\sigma M : B$.

The case for a variable pattern uses the value substitution lemma above, and the remaining cases are straightforward. □

Theorem 6.4.4 (Progress and Type Preservation)
If $\vdash M : A$ *then one of the following holds.*

1. M *is a value.*
2. $M \mapsto N$, *with* $\vdash N : A$.
3. $M \mapsto$ *MatchException.*

Proof: (sketch)
By induction on the structure of the typing derivation for M, using the above substitution lemmas for values, expressions and patterns, and using inversion on the form of values inhabiting function, product, and base types. □

This theorem is not quite sufficient to ensure that programs never go wrong: it ensures that non-values always reduce to appropriately typed program states,, but does not rule the possibility of terms having other possible reductions that lead to ill-typed terms. To rule out this possibility, we now prove that evaluation is deterministic. First, we require the following lemma.

Lemma 6.4.5 *If N is not a value then $E[N]$ is not a value.*

Proof: By induction on the structure of E.

Case: $E = []$: Then, $E[N] = N$, hence is not a value.

Case: $E = c\,E_1$: Then $E_1[N]$ is not a value (ind. hyp.), so $c\,E_1[N]$ is not a value.

Case: $E = (E_1,M_2)$: Then $E_1[N]$ is not a value (ind. hyp.), so $(E_1[N],M_2)$ is not a value.

Case: $E = (V_1,E_2)$: Then $E_2[N]$ is not a value (ind. hyp.), so $(V_1,E_2[N])$ is not a value.

Case: $E = E_1\,M_2$ or $V_1\,E_2$ or **case** E_1 **of** Ω: Immediate.

□

Theorem 6.4.6 (Determinism of $\mapsto$)
If $M \mapsto \pi$ and $M \mapsto \pi'$ then $\pi = \pi'$.

Proof:
Let $E[X]$ and $E'[X']$ be the decompositions of M for the L.H.S. of the two reductions. Then, X and X' are each one of the following forms.

$$X, X' \quad ::= \quad (\lambda x.N)\,V \mid \textbf{case}\ V\ \textbf{of}\ \Omega \mid \textbf{fix}\,u.N$$

Hence, neither X nor X' is a value.

We show by induction on the structure of M that $M = E[X] = E[X']$ implies that $E = E'$ and $X = X'$. The result follows because the $E[X]$ on the L.H.S. of the reduction rules have disjoint possibilities for X, hence $M \mapsto \pi$ and $M \mapsto \pi'$ must be obtained by the same rule, and the R.H.S. of each reduction rule is completely determined by the choice of E and X.

We have the following cases for $M = E[X] = E'[X']$ in the induction.

Case: $M = x$ or $\lambda x.M$ or cV or (V_1,V_2) or $()$.

Cannot occur, since $M = E[X]$ and X not a value implies that
M is not a value, by the preceding lemma.

Case: $M = M_1\,M_2$ with M_1 not a value.

Then the only possibility is $E[X] = E_1[X]\ M_2$ and $E'[X'] = E'_1[X']\ M_2$.
But then $E_1[X] = E'_1[X'] = M_1$.
And then $E_1 = E'_1$ and $X = X'$ (ind. hyp.).
Thus, $E = E'$.

Case: $M = V_1\,M_2$ with M_2 not a value.

Then the only possibility is $E[X] = V_1\ E_2[X]$ and $E'[X'] = V_1\ E'_2[X']$
(using the above lemma to rule out $E[X] = E_1[X]\,M_2$ and $E[X'] = E'_1[X']\,M_2$).

But then $E_2[X] = E_2'[X'] = M_2$.
And then $E_2 = E_2'$ and $X = X'$ (ind. hyp.).
Thus, $E = E'$.

Case: $M = V_1\, V_2$.
Then the only possibility is $V_1 = \lambda x.N_1$
and $E = []$ with $X = (\lambda x.N_1)\, V_2$ and $E' = []$ with $X' = (\lambda x.N_1)\, V_2$.

Case: $M = c\, M_1$ with M_1 not a value.
Then the only possibility is $E[X] = c\, E_1[X]$ and $E'[X'] = c\, E_1'[X']$.
But then $E_1[X] = E_1'[X'] = M_1$.
And then $E_1 = E_1'$ and $X = X'$ (ind. hyp.).
Thus, $E = E'$.

Case: $M = (M_1,M_2)$ with M_1 not a value.
Then the only possibility is $E[X] = (E_1[X],M_2)$ and $E'[X'] = (E_1'[X'],M_2)$.
But then $E_1[X] = E_1'[X'] = M_1$.
And then $E_1 = E_1'$ and $X = X'$ (ind. hyp.).
Thus, $E = E'$.

Case: $M = (V_1,M_2)$ with M_2 not a value.
Then the only possibility is $E[X] = (V_1,E_2[X])$ and $E'[X'] = (V_1,E_2'[X'])$
(using the above lemma to rule out $E[X] = (E_1[X],M_2)$ and $E[X'] = (E_1'[X'],M_2)$).
But then $E_2[X] = E_2'[X'] = M_2$.
And then $E_2 = E_2'$ and $X = X'$ (ind. hyp.).
Thus, $E = E'$.

Case: $M = \mathbf{case}\ M_1\ \mathbf{of}\ \Omega$ with M_1 not a value.
Then the only possibility is $E[X] = \mathbf{case}\ E_1[X]\ \mathbf{of}\ \Omega$ and $E'[X'] = \mathbf{case}\ E_1'[X']\ \mathbf{of}\ \Omega$.
But then $E_1[X] = E_1'[X'] = M_1$.
And then $E_1 = E_1'$ and $X = X'$ (ind. hyp.).
Thus, $E = E'$.

Case: $M = \mathbf{case}\ V_1\ \mathbf{of}\ \Omega$.
Then the only possibility is $E = []$ with $X = \mathbf{case}\ V_1\ \mathbf{of}\ \Omega$
and $E' = []$ with $X' = \mathbf{case}\ V_1\ \mathbf{of}\ \Omega$.

Case: $M = u$.
Cannot occur: no decomposition $E[X]$ matches the form u.

Case: $M = \mathbf{fix}\, u.M_1$.
Then the only possibility is $E = []$ with $X = \mathbf{fix}\, u.M_1$ and $E' = []$ with $X' = \mathbf{fix}\, u.M_1$.

□

We note that it is necessary include match exceptions in the semantics in order for the progress theorem to hold. By contrast, the corresponding result for sorts rules out match exceptions,

and would thus satisfy progress with respect to a semantics that omitted match exceptions. However, since we take the view that types judge the validity of terms, and not sorts, the semantics with match exceptions is the one that we are most interested in.

The semantics without match exceptions may still be of interest though: we can consider it to be a "refinement" of the semantics that includes match exceptions. It seems likely that there are other instances where refinements of the type system of a language can be related to such "refined" semantics. We plan to consider this notion of "refining" semantics further in future work.

6.5 Declarative sort assignment

We now consider sort assignment for $ML^{\&\mathbf{case}}$. The sort assignment judgments require an appropriate form of pattern sort context Δ containing assumptions for both variables and patterns, which is defined below. For simplicity, value variables x are treated as a special case of patterns, hence we have two kinds of assumptions: those for patterns and those for expressions variables u. Pattern sort assumptions have the form $P{\in}Z$ where Z is a *pattern sort*, which are a generalization of sorts that are designed to accurately describe the values that may reach a particular branch of a **case** expression. The form $P{\in}Z$ can thus be used to express an assumption that some of the values reaching a branch match the branch's pattern. Pattern sort contexts Δ thus generalize sort contexts with features appropriate for a language with pattern matching. The new features may only appear at the top-level of a pattern sort, i.e., outside of all sorts R that appear within the pattern sort.

$$\begin{array}{rrcl} \text{Pattern Sort} & Z & ::= & R \mid c\,Z \mid (Z_1, Z_2) \mid () \mid Z_1 \sqcup Z_2 \mid \bot^A \\ \text{Pattern Sort Context} & \Delta & ::= & \cdot \mid \Delta, P{\in}Z \mid \Delta, u{\in}R \end{array}$$

The constructs $c\,Z$, $(Z_1{,}Z_2)$ and $()$ mirror the constructs for values and patterns. An alternative would be to use the notation $Z_1 \times Z_2$ and $\mathsf{1}$ instead of $(Z_1{,}Z_2)$ and $()$, but this would conflict with the notation for sorts, in particular when Z_1 and Z_2 are sorts, potentially leading to confusion. Further, the purpose of these constructs is to directly describe the structure of values, so it seems preferable to mirror the constructs used for values (and in the case of $c\,Z$ there seems to be no obvious alternative). Informally, $c\,Z$ is the pattern sort containing every value $c\,V$ that has V contained in Z, $(Z_1{,}Z_2)$ contains every value $(V_1{,}V_2)$ that has V_1 in Z_1 and V_2 in Z_2, and the pattern sort $()$ contains only the value $()$.

The construct $Z_1 \sqcup Z_2$ forms a union of two pattern sorts, i.e., a pattern sort containing all values in either Z_1 or Z_2. The construct $\bot^A$ is an empty pattern sort. The type A in $\bot^A$ is required so that each pattern sort refines a unique type. We omit A when it is clear from context, or of little interest.

We extend notion of refinement to pattern sorts and pattern sort contexts as follows.

$\vdash_\Sigma Z \boxdot A$ $\quad$ Z is a valid pattern sort refining type A.

$\vdash_\Sigma \Delta \sqsubset \Gamma; \Phi$ $\quad$ Δ is a valid pattern sort context refining Γ and Φ.

Valid pattern sorts

$$\frac{R \sqsubset A}{R \boxdot A} \qquad \frac{Z \boxdot B \quad cB \text{ in } D \quad a = D \text{ in } \Sigma}{cZ \boxdot a}$$

$$\frac{Z_1 \boxdot A \qquad Z_2 \boxdot B}{(Z_1,Z_2) \boxdot A \times B} \qquad \frac{}{() \boxdot 1} \qquad \frac{Z_1 \boxdot A \qquad Z_2 \boxdot A}{Z_1 \sqcup Z_2 \boxdot A} \qquad \frac{}{\bot^A \boxdot A}$$

Valid pattern sort contexts

$$\frac{}{\cdot \sqsubset \cdot} \qquad \frac{\Delta \sqsubset \Gamma; \Phi \qquad Z \boxdot A}{(\Delta, P{\in}Z) \sqsubset \Gamma; (\Phi, P{:}A)} \qquad \frac{\Delta \sqsubset \Gamma; \Phi \qquad R \sqsubset A}{(\Delta, u{\in}R) \sqsubset (\Gamma, u{:}A); \Phi}$$

For uniqueness, the refinement judgment for pattern sort contexts $\Delta \sqsubset \Gamma; \Phi$ places all assumptions of the form $x{:}A$ in the pattern context Φ, although they could equally be placed in Γ. This is not so important, since $\Gamma; (\Phi, x{:}A) \vdash M{:}A$ if and only if $(\Gamma, x{:}A); \Phi \vdash M{:}A$. In some cases we need to relate a pattern context Φ to an ordinary context Γ which includes appropriate types to the variables in the patterns in Φ. We do this via the following judgment.

$\vdash_\Sigma \Phi \cong \Gamma$ Pattern context Φ is equivalent to Γ

Equivalence of pattern and ordinary contexts

$$\frac{}{\cdot \cong \cdot} \qquad \frac{\Phi \cong \Gamma_1 \qquad \Gamma_2 \vdash P : A \qquad \text{every } x \text{ in } \Gamma_2 \text{ is in } P}{\Phi, P{:}A \cong \Gamma_1, \Gamma_2}$$

The condition "every x in Γ_2 is in P" is required since the typing derivation allows unused variables in Γ_2. An equivalent alternative that avoids this condition would be to use left rules to relate Φ and Γ, via a hypothetical judgment that is parametric in the M and A on the right, as follows.

$$\frac{\begin{array}{c}[\Gamma; \cdot \vdash M : A] \\ \vdots \\ \cdot; \Phi \vdash M : A\end{array}}{\Phi \cong \Gamma}$$

We now define the sort assignment judgments. The first is similar to the standard sort assignment judgment as in previous chapters, although generalized to pattern sort contexts. The second is used when matching against the body of a **case** expression.

$\Delta \vdash_\Sigma M \in R$	Expression M has sort R under pattern sort context Δ and signature Σ. (Where $\Gamma; \Phi \vdash_\Sigma M \in A$ with $\Delta \sqsubset \Gamma; \Phi$ and $R \sqsubset A$.)
$\Delta \vdash_\Sigma Z \bowtie \Omega \in R$	Matching pattern sort Z with branches Ω yields sort R under pattern sort context Δ and signature Σ. (Where $\Gamma_1, \Gamma_2 \vdash A \bowtie \Omega : B$ with $\Delta \sqsubset \Gamma_1; \Phi_2$ and $\Phi_2 \cong \Gamma_2$ and $Z \sqsubset A$ and $R \sqsubset B$.)

We also require a judgment which defines an inclusion of pattern sorts in sort unions, and is used to ensure that assumptions for variables of the form $x{\in}Z$ can be converted into ordinary sort assumptions $x{\in}R$ (via a case analysis when the union has many components). A sort union Y is a union of a list of sorts. Formally, we define sort unions as a subclass of pattern sorts, as follows.

$$\text{Sort Union} \quad Y \quad ::= \quad \bot \mid Y \sqcup R$$

The inclusion of pattern sorts in sort unions is defined as follows.

$\vdash_\Sigma Z \preceq Y$	Pattern sort Z is contained in sort union Y under Σ. (Where $Z \boxdot A$ and $Y \boxdot A$ for some A.)

We generally omit the signature Σ from instances of the above judgments to avoid clutter, since it is fixed throughout the rules which follow. We also often omit the $\vdash$ when there are no assumptions, and in particular we write $Z \preceq Y$ for $\vdash_\Sigma Z \preceq Y$.

The sort assignment rules appear in Figure 6.2, and the pattern sort inclusion rules appear in Figure 6.3. We divide the sort assignment rules into three parts, for consistency with the presentation of the typing rules, even though here the left rules are part of the same judgment as the ordinary, right rules for term constructs.

The sort assignment and pattern sort inclusion rules make use of the definitions of constructor sorts and inversion principles from Section 6.1, which in turn make use of the definitions of datasort bodies from Section 5.5. For convenience, these definitions are repeated in Figure 6.4.

Right rules

One critical feature of the sort assignment rules is that the variable rule is restricted to assumptions involving ordinary sorts $x{\in}R$, and does not allow other forms of pattern sorts. Thus, pattern sorts are only used during pattern matching, and for assumptions for variables bound in patterns, but only ordinary sorts are assigned to terms.

Otherwise, the first group of rules is essentially the rules from Chapter 3 other than the rule for constants, which is replaced by a rule for constructor applications using the form $c\,S \rightarrowtail \rho$ introduced in Section 6.1. We add standard introduction rules for products, rules for **fix** and expression variables based on those in Chapter 4, and a rule for **case** expressions which matches a sort for the case object with the cases in the **case** expression.

$$\frac{x{\in}R \text{ in } \Delta}{\Delta \vdash x \in R} \qquad \frac{\Delta, x{\in}R \vdash M \in S}{\Delta \vdash \lambda x.M \in R \to S} \qquad \frac{\Delta \vdash M \in R \to S \qquad \Delta \vdash N \in R}{\Delta \vdash M\, N \in S}$$

$$\frac{u{\in}R \text{ in } \Delta}{\Delta \vdash u \in R} \qquad \frac{\Delta, u{\in}R \vdash M \in R}{\Delta \vdash \mathbf{fix}\, u.M \in R} \qquad \frac{c\, S \rightarrowtail \rho \qquad \Delta \vdash M \in S}{\Delta \vdash c\, M \in \rho}$$

$$\frac{\Delta \vdash M \in R \qquad \Delta \vdash N \in S}{\Delta \vdash (M, N) \in R{\times}S} \qquad \frac{}{\Delta \vdash () \in 1}$$

$$\frac{\Delta \vdash M \in R \qquad \Delta \vdash R \bowtie \Omega \in S}{\Delta \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S} \qquad \frac{\Delta \vdash M \in R \quad R \trianglelefteq S}{\Delta \vdash M \in S}$$

$$\frac{\Delta \vdash V \in R \qquad \Delta \vdash V \in S}{\Delta \vdash V \in R \& S} \qquad \frac{}{\Delta \vdash V \in \top^A}$$

. . . . . .

$$\frac{\Delta, P{\in}Z \vdash M{\in}R \quad \Delta \vdash (Z \backslash P) \bowtie \Omega \in R}{\Delta \vdash Z \bowtie (P \Rightarrow M \mid \Omega) \in R} \qquad \frac{Z \preceq \bot}{\Delta \vdash Z \bowtie \cdot \in R}$$

. . . . . .

$$\frac{\Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash M{\in}R}{\Delta, (P_1, P_2) \in (Z_1, Z_2) \vdash M{\in}R} \qquad \frac{\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash M{\in}R}{\Delta, (P_1, P_2) \in S_1{\times}S_2 \vdash M{\in}R}$$

$$\frac{\Delta \vdash M{\in}R}{\Delta, () \in () \vdash M{\in}R} \qquad \frac{\Delta \vdash M{\in}R}{\Delta, () \in 1 \vdash M{\in}R}$$

$$\frac{\Delta, P{\in}Z \vdash M{\in}R}{\Delta,\, c\, P \in c\, Z \vdash M{\in}R} \qquad \frac{c \neq c_2}{\Delta,\, c\, P \in c_2\, Z \vdash M{\in}R} \qquad \frac{\Delta,\, c\, P \in \mathsf{inv}(\rho) \vdash M{\in}R}{\Delta,\, c\, P \in \rho \vdash M{\in}R}$$

$$\frac{}{\Delta, P{\in}\bot \vdash M{\in}R} \qquad \frac{\Delta, P{\in}Z_1 \vdash M{\in}R \qquad \Delta, P{\in}Z_2 \vdash M{\in}R}{\Delta, P \in Z_1 \sqcup Z_2 \vdash M{\in}R}$$

$$\frac{Z \preceq Y \qquad \Delta, x{\in}Y \vdash M{\in}R}{\Delta, x{\in}Z \vdash M{\in}R} \qquad \frac{S \preceq Y \qquad \Delta, P{\in}Y \vdash M{\in}R}{\Delta, P{\in}S \vdash M{\in}R}$$

Figure 6.2: Sort assignment rules for terms, branches, and patterns

$$\frac{\vdash_\Sigma R \trianglelefteq S}{R \preceq S} \qquad \frac{\vdash_\Sigma R \trianglelefteq \cdot}{R \preceq \bot} \qquad \frac{Z_1 \preceq Y_1 \qquad Z_2 \preceq Y_2}{Z_1 \sqcup Z_2 \preceq \sqcup\{R \mid R \text{ in } Y_1 \text{ or } Y_2\}}$$

$$\frac{}{() \preceq 1} \qquad \frac{Z_1 \preceq Y_1 \qquad Z_2 \preceq Y_2}{(Z_1, Z_2) \preceq \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}} \qquad \frac{}{\bot \preceq \bot}$$

$$\frac{Z \preceq R_1 \sqcup \ldots \sqcup R_n \qquad c\,R_1 \rightarrowtail \rho_1 \;\; \ldots \;\; c\,R_n \rightarrowtail \rho_n}{c\,Z \preceq \rho_1 \sqcup \ldots \sqcup \rho_n}$$

Figure 6.3: Pattern sort inclusion rules

$$\begin{aligned}
\mathsf{body}(r) &= \Psi && \text{when } r = \Psi \text{ in } \Sigma \\
\mathsf{body}(\rho_1 \& \rho_2) &= \sqcup\{c\,(R_1 \& R_2) \mid c\,R_1 \text{ in } \mathsf{body}(\rho_1), c\,R_2 \text{ in } \mathsf{body}(\rho_2)\} \\
\mathsf{body}(\top^a) &= \sqcup\{c \top^A \mid c\,A \text{ in } D\} && \text{when } a = D \text{ in } \Sigma \\
\mathsf{inv}(\rho) &= \mathsf{body}(\&\{r \mid \rho \trianglelefteq r\})
\end{aligned}$$

$\vdash_\Sigma c\,S \rightarrowtail \rho$ Constructor c maps from S to ρ.

$$\frac{c\,R \text{ in } \mathsf{body}(\rho) \qquad S \trianglelefteq R}{\vdash_\Sigma c\,S \rightarrowtail \rho}$$

Figure 6.4: Bodies, inversion principles and constructor sorts (repeated from before)

Rules for lists of branches

The judgment for the branches of a **case** expression considers the body of each branch under the assumption that the pattern for the branch matches the pattern sort describing the values that may reach that case. It uses the subtraction operator $Z \backslash P$ (defined below) to calculate a pattern sort describing the values that reach each branch. When there are no more branches, we check that the final pattern sort is empty, ensuring that every possible value is matched by some case.

Left rules

The third group contains the left rules for the judgment $\Delta \vdash_\Sigma M \in R$. These rules match patterns against pattern sorts in the pattern assumptions $P{\in}Z$ in Δ. We include enough rules to match every well-formed pattern assumption $P{\in}Z$ in the conclusion. We allow the order of assumptions in Δ to be exchanged, hence these rules apply to any assumption in the context Δ, not just the last one. We could make this explicit in the rules by adding a Δ' to the end of each context. E.g., the rule for $\sqcup$ could be written as follows.

$$\frac{\Delta, P{\in}Z_1, \Delta' \vdash M{\in}R \qquad \Delta, P{\in}Z_2, \Delta' \vdash M{\in}R}{\Delta, P \in Z_1 \sqcup Z_2, \Delta' \vdash M{\in}R}$$

Omitting these Δ' and implicitly allowing exchange results in a less cluttered presentation. This is important since already each rule is parametric in a Δ, an M and an R; with Δ' as well the underlying simplicity of each rule would be less obvious. Further, these declarative rules are intended to reflect the reasoning used by programmer, and allowing implicit exchange seems to most accurately reflect the intention that the programmer need not keep track of the order of assumptions.

The left rule for an assumption of the form $c\,P{\in}\rho$ expands ρ using its inversion principle $\mathsf{inv}(\rho)$ as defined in Section 6.1. Note that an inversion principle has the form of a datasort body $\Psi = c_1\,R_1 \sqcup \ldots \sqcup c_n\,R_n$, which corresponds to a pattern sort.

Inversion principles involve unions, and unions also arise from the subtraction operator $Z \backslash P$ when P involves pairs. The left rule for a pattern assumption involving such a union $P{\in}Z_1 \sqcup Z_2$ performs a case analysis: we check that the desired conclusion holds under $P{\in}Z_1$ and under $P{\in}Z_2$. This rule is essentially the left rule for disjunction, and our operator $\sqcup$ is essentially a restricted form of union types designed specifically to allow case analysis when assigning sorts to pattern matches. Using left rules allows us to specify this case analysis in a very natural way, while other approaches to sort assignment for patterns do not seem well suited for this purpose.

Two of the left rules involve pattern assumptions that are false: $c\,P{\in}c_2\,Z$ and $P{\in}\bot$. In these cases the conclusion holds without checking the expression M at all. These correspond to situations where the expression M appears in a branch that is unreachable.

Finally, we have two forms of subsumption: the first applies only to variable pattern assumptions $x{\in}Z$, while the second applies only to pattern sort assumptions with (non-pattern) sorts $P{\in}S$. Each allows the pattern sort to be replaced by a sort union Y. Attempting to combine the two left subsumption rules into a single, more general rule leads to a situation

where an inversion principle can be applied inappropriately via subsumption from $cP{\in}cZ$ to $cP{\in}Y$, with $Y = \rho_1 \sqcup \ldots \sqcup \rho_n$. [3]

Inclusion in sort unions

The rules for the judgment $Z \preceq Y$ are designed to be somewhat minimal, while still incorporating subsorting $R \trianglelefteq S$ and allowing every pattern sort to be replaced by a sort union. It seems possible to enrich this judgment to a more general inclusion between pattern sorts, although it is not clear what we would gain from doing so, and it is clear that we need to be careful with such an enrichment. In particular, we do not want to allow every inclusion that is validated by considering the sets of values that inhabit pattern sorts. If we did, it would allow unrestricted case analysis of a datasort to be performed, since we would have $\rho \preceq c_1\, R_1 \sqcup \ldots \sqcup c_n\, R_n$ provided all values in ρ have the form $c_i V$ with $V \in R_i$ (thus including $\rho \preceq \mathsf{inv}(\rho)$). This violates one of the principles behind the design of the system: that case analysis for a datasort is only performed when it is matched against a constructor, and only using its inversion principle.

Pattern sort subtraction

The rule for matching a pattern sort against a case makes use of the subtraction operator $Z \backslash P$ which calculates a pattern sort representing that part of Z not matched by the pattern P. The definition of this operator is closely based on the left rules for sort assignment: each rule involving a pattern assumption of the form $P{\in}Z$ leads to a clause for the definition of $Z \backslash P$. The definition of $Z \backslash P$ thus contains the following clauses. When Z and P match more than one clause we use the earliest one (although this choice is not significant).

$$
\begin{array}{rcll}
Z \setminus x & = & \bot & \\
(Z_1, Z_2) \setminus (P_1, P_2) & = & ((Z_1 \backslash P_1), Z_2) \sqcup (Z_1, (Z_2 \backslash P_2)) & \\
R \setminus (P_1, P_2) & = & ((R_1 \backslash P_1), R_2) \sqcup (R_1, (R_2 \backslash P_2)) & (\text{if } R \stackrel{\rightrightarrows}{\simeq} R_1 \otimes R_2) \\
Z \setminus () & = & \bot & \\
(c\,Z) \setminus (c\,P) & = & c\,(Z \backslash P) & \\
(c\,Z) \setminus (c'\,P) & = & c\,Z & (\text{when } c \neq c') \\
\rho \setminus (c\,P) & = & \mathsf{inv}(\rho) \setminus (c\,P) & \\
\bot \setminus P & = & \bot & \\
(Z_1 \sqcup Z_2) \setminus P & = & (Z_1 \backslash P) \sqcup (Z_2 \backslash P) &
\end{array}
$$

This definition is inductive, but requires a complicated well-ordering due to the case which uses the inversion principle $\mathsf{inv}(\rho)$. To simplify the situation, we formally make the definition inductive on P, Z lexicographically by modifying the right-hand side for this case to the slightly less intuitive expression which follows.

$$
\begin{array}{rcl}
\rho \setminus (c\,P) & = & \bigsqcup \{ c(R \backslash P) \mid c\,R \text{ in } \mathsf{inv}(\rho) \} \\
& \sqcup & \bigsqcup \{ c'\,R' \mid c'\,R' \text{ in } \mathsf{inv}(\rho) \text{ and } c' \neq c \}
\end{array}
$$

[3] *Previously these were combined into one form, but this leads to the completeness theorem for the checking algorithm being false.*

The definition of $Z \backslash P$ covers every case that can arise when the types associated with Z and P are the same. Each clause is relatively straightforward. The result of subtracting a pair pattern is a union: a pair fails to match a pair pattern when either the first component fails to match, or the second component fails to match. To subtract a constructor pattern $c\,P$ from a base refinement ρ we first expand ρ using its inversion principle. This follows the guiding principle that we make use of the inversion principle for ρ whenever an object with sort ρ is matched against a constructor, and do not use it at any other time.

One notable choice that we have made here is in the definition of subtraction for products. An alternative definition is the following (and similarly for $R \backslash (P_1, P_2)$).

$$\begin{array}{rcl} (Z_1, Z_2) \setminus (P_1, P_2) & = & (Z_1 \backslash P_1,\ Z_2 \backslash P_2) \sqcup (Z_1 \backslash P_1,\ Z_2 \wedge P_2) \\ & & \qquad \sqcup\ (Z_1 \wedge P_1,\ Z_2 \backslash P_2) \end{array}$$

This requires the intersection operation on pattern sorts $Z \wedge P$ to be defined, in a similar manner to $Z \backslash P$. We have chosen against this alternative since it is a little more complex, and we would like the declarative sort assignment system to be as simple as possible.

However, we observe that the above alternative definition leads to a slightly more detailed case analysis. We have implemented and experimented with this alternative definition in our sort checker for SML, and so far we have not encountered an example where this more detailed analysis is desirable. Interestingly, the more detailed alternative is often more efficient: when there is a sequence of subtractions for products the actual definition often leads to unions with many redundant components because values in $((Z_1 \backslash P_1), (Z_2 \backslash P_2))$ are represented in both parts of the union. Such redundant components are removed prior to checking the body of a branch, but they still have a measurable effect on the performance of the implementation.

6.6 Progress and sort preservation theorem

We now consider a proof of a theorem that states that our operational semantics preserves sorts, and additionally that progress is satisfied without the possibility of a match exception for closed terms which can be assigned sorts. In order to prove this theorem we require a suitable notion of when closed values inhabit pattern sorts: previously pattern sorts have only been considered in assumptions. We thus introduce a new judgment for this purpose, with the expected rules. The syntax for this judgment $V \in Z$ intersects with the abbreviated syntax for a sort assignment judgment $M \in R$ with no assumptions. However no confusion should occur, since when the pattern sort Z is a sort S, the two judgments coincide. When we wish to explicitly refer to the original sort assignment judgment, we include the prefix $\vdash$ (as in the first rule below).

$V \in Z$ Value V is assigned sort Z.
(Where $V{:}A$ and $Z \sqsubset A$.)

$$\frac{\vdash V \in R}{V \in R} \qquad \frac{}{() \in ()} \qquad \frac{V_1 \in Z_1 \qquad V_2 \in Z_2}{(V_1{,}V_2) \in (Z_1{,}Z_2)}$$

$$\frac{V \in Z}{cV \in cZ} \qquad \frac{V \in Z_1}{V \in Z_1 \sqcup Z_2} \qquad \frac{V \in Z_2}{V \in Z_1 \sqcup Z_2}$$

Our first two lemmas correspond directly to those in the proof of the progress and type preservation theorem in Section 4.5. The first is value preservation, which is extended to the value substitutions $\sigma = \{V_1/x_1, \ldots, V_n/x_n\}$ used in the semantics for pattern matching, and which include $\{V'/x\}V$ as a special case.

Lemma 6.6.1 (Value Preservation)

1. *σV is a value.*
2. *$\{M/u\}V$ is a value.*

Proof: By straightforward inductions on V. The case for a variable that appears in σ follows because σ only contains values. The case for u cannot occur, since u is not a variable. The remaining cases simply rebuild the value. □

The expression substitution lemma for this language has two parts, corresponding to the two sort assignment judgments. We remark that patterns bind variables, so substitution avoids capturing such variables. We delay consideration of the value substitution lemma until later, when we will be in a position to prove an appropriate generalization for pattern sort assumptions.

Lemma 6.6.2 (Expression Substitution Lemma)
If $\Delta \vdash M : R$ then the following hold.

1. *If $\Delta, u{:}R \vdash N : S$ then $\Delta \vdash \{M/u\}N : S$.*
2. *If $\Delta, u{:}R \vdash Z \bowtie \Omega \in S$ then $\Delta \vdash Z \bowtie \{M/u\}\Omega \in S$.*

Proof: By induction on the structure of the derivation involving u. The case for the variable rule with u makes use of the derivation of $\Delta \vdash M : R$. The remaining cases simply rebuild the derivation. □

We now prove some lemmas that characterize the values inhabiting sorts, pattern sorts, sort unions Y, the sort unions U of Chapter 5, and datasort bodies.

Lemma 6.6.3 (Weak Value Inversion for Datasorts)
If $V{\in}\rho$ then $V = cW$ and $W{\in}R$
with cR in $\mathsf{body}(\rho')$ for some $\rho' \trianglelefteq \rho$.

Proof: By induction on the structure of $\mathcal{D} :: V{\in}\rho$.

Case: $\dfrac{c\,S \rightarrowtail \rho \qquad \vdash W \in S}{\vdash c\,W \in \rho}$

cR in $\mathsf{body}(\rho)$ with $S \trianglelefteq R$	Inv. on $\rightarrowtail$
$W{\in}R$	Subsumption Rule
$\rho \trianglelefteq \rho$	Reflexivity (5.8.11)

Case: $\dfrac{\vdash V \in \rho_2 \qquad \rho_2 \trianglelefteq \rho_1}{\vdash V \in \rho_1}$

$V = cW$ and $W{\in}R$ with	
cR in $\mathsf{body}(\rho_3)$ and $\rho_3 \trianglelefteq \rho_2$.	Ind. Hyp.
$\rho_3 \trianglelefteq \rho_1$	Transitivity (5.8.11)

Case: $\dfrac{\vdash V \in \rho_1 \qquad \vdash V \in \rho_2}{\vdash V \in \rho_1 \,\&\, \rho_2}$

$V = cW$ and $W{\in}R_1$	
with cR_1 in $\mathsf{body}(\rho_1')$ and $\rho_1' \trianglelefteq \rho_1$.	Ind. Hyp.
$V = cW$ and $W{\in}R_2$	
with cR_2 in $\mathsf{body}(\rho_2')$ and $\rho_2' \trianglelefteq \rho_2$.	Ind. Hyp.
$W{\in}R_1\&R_2$	&-Intro
$c\,(R_1\&R_2)$ in $\mathsf{body}(\rho_1'\&\rho_2')$	Def. body
$\rho_1'\&\rho_2' \trianglelefteq \rho_1\&\rho_2$	Lemma 5.8.11

Case: $\dfrac{}{\vdash V \in \top^a}$

$V : a$	Well-formedness of $V \in \top^a$
$V = c\,W$ with $W : B$	
and $(c\,B)$ in D and $(a = D)$ in Σ	Inv. on $V : a$
$W \in \top^B$	$\top$-Intro
$c\top^B$ in $\mathsf{body}(\top^a)$	Def. body

□

Lemma 6.6.4 *If* $\vdash (V_1, V_2) \in R$
then $\vdash V_1 \in R_1$ *and* $\vdash V_2 \in R_2$ *and* $R_1{\times}R_2 \trianglelefteq R$ *for some* R_1, R_2.

Proof: By induction on $\mathcal{D} :: \vdash V \in R$.

Case: $\dfrac{\vdash V_1 \in R_1 \qquad \vdash V_2 \in R_2}{\vdash (V_1, V_2) \in R_1{\times}R_2}$

Then $R = R_1{\times}R_2$,

and $R_1 \times R_2 \trianglelefteq R_1 \times R_2$ Reflexivity (5.8.11)

Case: $\dfrac{\vdash (V_1, V_2) \in S \qquad S \trianglelefteq R}{\vdash (V_1, V_2) \in R}$

$V_1 \in S_1$ and $V_2 \in S_2$ and $S_1 \times S_2 \trianglelefteq S$ Ind. Hyp.

$S_1 \times S_2 \trianglelefteq R$ Transitivity (5.8.11)

Case: $\dfrac{\vdash (V_1, V_2) \in R \qquad \vdash (V_1, V_2) \in S}{\vdash (V_1, V_2) \in R \,\&\, S}$

$V_1 \in R_1$ and $V_2 \in R_2$ and $R_1 \times R_2 \trianglelefteq R$ Ind. Hyp.

$V_1 \in S_1$ and $V_2 \in S_2$ and $S_1 \times S_2 \trianglelefteq S$ Ind. Hyp.

$V_1 \in R_1 \& S_1$ and $V_2 \in R_2 \& S_2$ &-Intro

$(R_1 \& S_1) \times (R_2 \& S_2) \trianglelefteq R$ Lemmas 5.8.11, 5.8.10

$(R_1 \& S_1) \times (R_2 \& S_2) \trianglelefteq S$ Lemmas 5.8.11, 5.8.10

$(R_1 \& S_1) \times (R_2 \& S_2) \trianglelefteq R \& S$ Lemma 5.8.11

Case: $\dfrac{}{\vdash (V_1, V_2) \in \top^{A \times B}}$

$V_1 \in \top^A$ and $V_2 \in \top^B$ $\top$-Intro

$\top^A \times \top^B \trianglelefteq \top^{A \times B}$ Lemma 5.8.10

□

Lemma 6.6.5 *If $R \sqsubset \mathbf{1}$ then $\cdot \vdash () \in R$.*

Proof: By a straightforward induction on R.
The cases for $R = \mathbf{1}$ and $R = \top^{\mathbf{1}}$ are immediate.
The case for $R = R_1 \,\&\, R_2$ applies the induction hypothesis to R_1 and R_2, then uses &-introduction. □

Lemma 6.6.6 (Product Inclusion Inversion)
If $R_1 \times R_2 \trianglelefteq S_1 \times S_2$ then one of the following holds:

1. *$R_1 \trianglelefteq S_1$ and $R_2 \trianglelefteq S_2$*
2. *$R_1 \trianglelefteq \cdot$*
3. *$R_2 \trianglelefteq \cdot$*

Proof: By inversion on $\mathcal{D} :: R_1 \times R_2 \trianglelefteq S_1 \times S_2$.

$(R_1 \backslash \cdot) \otimes (R_2 \backslash \cdot) \trianglelefteq S_1 \times S_2$	Inv.
$(R_1 \backslash S_1) \otimes (R_2 \backslash \cdot) \trianglelefteq \cdot$	Inv.
$(R_1 \backslash \cdot) \otimes (R_2 \backslash S_2) \trianglelefteq \cdot$	Inv.
$R_1 \trianglelefteq S_1$ or $R_2 \trianglelefteq \cdot$	Inv.
$R_1 \trianglelefteq \cdot$ or $R_2 \trianglelefteq S_1$	Inv.
$(R_1 \trianglelefteq S_1$ and $R_2 \trianglelefteq S_2)$ or $R_1 \trianglelefteq \cdot$ or $R_2 \trianglelefteq \cdot$	

□

We now show a generalized form of subsumption, which holds trivially for inclusions of the form $R \trianglelefteq S$, by the subsumption rule. However, in its general form, including unions and datasort bodies, this is the key lemma which allows us to determine the impossibility of values inhabiting some pattern sorts (such as at the end of a list of branches), and also allows us to justify our inversion principles.

This lemma has similarities to the soundness theorem for $\trianglelefteq$ (5.6.10 and 5.8.2): both relate the inhabitants of sorts, unions, and bodies when an algorithmic inclusion holds. However, here the notion of inhabitation is via the sort assignment judgment, while in the soundness theorem inhabitation was via the inductive semantics. The proofs of the two lemmas have a similar structure.

Lemma 6.6.7 (Generalized subsumption)

1. *If* $R \trianglelefteq U$ *and* $V \in R$
   then $V \in S$ *for some* S *in* U.

2. *If* $\Psi_1 \trianglelefteq \Psi_2$ *and* $V \in R$ *and* $c\,R$ *in* Ψ_1
   then $V \in S$ *with* $c\,S$ *in* Ψ_2.

3. *If* $R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq U$
   and $V_1 \in R_1$ *and* $V_2 \in R_2$ *and* $V = (V_1, V_2)$
   then $V_1 \in S_1$ *for some* S_1 *in* U_1
   or $V_2 \in S_2$ *for some* S_2 *in* U_2
   or $V \in S$ *for some* S *in* U.

Proof: By induction on the structure of the value V and the subsorting derivation, lexicographically. We have the following cases for the derivation.

Case: $\dfrac{\lfloor \rho \rfloor \trianglelefteq \lfloor u \rfloor \text{ in } \Theta}{\Theta \vdash \rho \trianglelefteq u}$

Cannot occur: $\Theta = \cdot$ throughout the lemma.

Case: $\dfrac{\lfloor \rho \rfloor \leq \lfloor u \rfloor \text{ not in } \cdot \qquad \lfloor \rho \rfloor \leq \lfloor u \rfloor \vdash \mathsf{body} \lfloor \rho \rfloor \trianglelefteq \mathsf{ubody} \lfloor u \rfloor}{\cdot \vdash \rho \trianglelefteq u}$

$V \in \rho$	Assumption
$V = cW$ with $W \in R_1$ and cR_1 in $\mathsf{body}(\rho_1)$ and $\rho_1 \trianglelefteq \rho$	Weak Inv. Lem. (6.6.3)
$\rho_1 \trianglelefteq u$	Transitivity (5.8.11)
$\mathsf{body}(\rho_1) \trianglelefteq \mathsf{body}(u)$	Lemma 5.6.2
$W \in S_1$ and cS_1 in $\mathsf{ubody}(u)$	Ind. Hyp. (W smaller)
cS_1 in $\mathsf{body}(\rho_1)$ for some ρ_1 in u	Def. ubody
$cS_1 \rightarrowtail \rho_1$	Rule for $\rightarrowtail$
$cW \in \rho_1$	Constr. App. Rule
As required, since $V = cW$ and ρ_1 in u.	

Case: $\dfrac{R \sqsubset 1}{\cdot \vdash R \trianglelefteq U \sqcup S}$

$V : 1$	Well-formedness of $V \in R$
$V = ()$	Inv. on $V : 1$
$S \sqsubset 1$	Well-formedness of $\cdot \vdash R \trianglelefteq U \sqcup S$
$V \in S$	Lemma 6.6.5

Case: $\dfrac{R \overset{\rightrightarrows}{\simeq} R_1 \otimes R_2 \qquad \cdot \vdash (R_1 \backslash \cdot) \otimes (R_2 \backslash \cdot) \trianglelefteq U}{\cdot \vdash R \trianglelefteq U}$

$V \in R$	Assumed
$R \sqsubset A_1 \times A_2$	Well-formedness of $R \overset{\rightrightarrows}{\simeq} R_1 \otimes R_2$
$V : A_1 \times A_2$	Well-formedness of $V \in R$
$V = (V_1, V_2)$	Inv. on $V : A_1 \times A_2$
$V_1 \in S_1$ and $V_2 \in S_2$ with $S_1 \times S_2 \trianglelefteq R$	Lemma 6.6.4
$R \trianglelefteq R_1 \times R_2$	Lemma 5.6.8
$S_1 \times S_2 \trianglelefteq R_1 \times R_2$	Transitivity (5.8.11)
($S_1 \trianglelefteq R_1$ and $S_2 \trianglelefteq R_2$) or $S_1 \trianglelefteq \cdot$ or $S_2 \trianglelefteq \cdot$	Prod. Inv. Lemma 6.6.6
If $S_1 \trianglelefteq \cdot$ then:	
$V_1 \in S_1$ with S_1 in $\cdot$	Ind. Hyp. (smaller V)
Contradiction.	
(The case for $S_2 \trianglelefteq \cdot$ is dual.)	
So, $S_1 \trianglelefteq R_1$ and $S_2 \trianglelefteq R_2$	
$V_1 \in R_1$ and $V_2 \in R_2$	Subsumption Rule
$(V_1, V_2) \in S$ with S in U	Ind. Hyp. (same V, subderivation)

Case: $\dfrac{}{\cdot \vdash \cdot \trianglelefteq \Psi}$

Cannot occur, since $\cdot$ is empty.

Case: $\dfrac{\cdot \vdash \Psi_1 \trianglelefteq \Psi_2 \qquad \cdot \vdash R_1 \trianglelefteq \sqcup\{S_2 \mid c_1\, S_2 \text{ in } \Psi_2\}}{\cdot \vdash \Psi_1 \sqcup c_1\, R_1 \trianglelefteq \Psi_2}$

$V \in R$ with $c\,R$ in $\Psi_1 \sqcup c_1\, R_1$ — Assumed

$c\,R$ in Ψ_1 or $c\,R = c_1\, R_1$

If $c\,R$ in Ψ_1 then:

$V \in S$ with $c\,S$ in Ψ_2 — Ind. Hyp. (same V, subderiv.)

Otherwise: $c = c_1$ and $R = R_1$

$V \in S$ with S in $\sqcup\{S \mid c\,S \text{ in } \Psi_2\}$ — Ind. Hyp. (same V, subderiv.)

$c\,S$ in Ψ_2

Case: $\dfrac{S \stackrel{\rightarrow}{\simeq} S_1 \otimes S_2 \qquad \begin{array}{l}\cdot \vdash (R_1 \backslash U_1 \sqcup S_1) \otimes (R_2 \backslash U_2) \trianglelefteq U \\ \cdot \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2 \sqcup S_2) \trianglelefteq U\end{array}}{\cdot \vdash (R_1 \backslash U_1) \otimes (R_2 \backslash U_2) \trianglelefteq U \sqcup S}$

Either $V_1 \in S_1'$ for some S_1' in $U_1 \sqcup S_1$

or $V_2 \in S_2'$ for some S_2' in U_2

or $V \in S'$ for some S' in U — Ind. Hyp. (same V, subderiv.)

Either $V_1 \in S_1''$ for some S_1'' in U_1

or $V_2 \in S_2''$ for some S_2'' in $U_2 \sqcup S_2$

or $V \in S''$ for some S'' in U — Ind. Hyp. (same V, subderiv.)

In each case, we have one of three required alternatives, except for the following case:

$V_1 \in S_1$ with S_1 in $U_1 \sqcup S_1$ and

$V_2 \in S_2$ with S_2 in $U_2 \sqcup S_2$

Then:

$(V_1, V_2) \in S_1 \times S_2$ — Rule for $\times$

$S_1 \times S_2 \trianglelefteq S$ — Lemmas 5.6.8, 5.8.10

$(V_1, V_2) \in S$ — Subsumption Rule

Case: $\dfrac{\cdot \vdash R_1 \trianglelefteq U_1}{\cdot \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq \cdot}$

$V_1 \in R_1$ — Assumed

$V_1 \in S_1$ for some S_1 in U_1 — Ind. Hyp. (V_1 smaller)

Case: $\dfrac{\cdot \vdash R_2 \trianglelefteq U_2}{\cdot \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq \cdot}$

Dual to the previous case.

Case: $\dfrac{\cdot \vdash S_1 \trianglelefteq R_1 \qquad \cdot \vdash R_2 \trianglelefteq S_2}{\cdot \vdash R_1 \to R_2 \trianglelefteq S_1 \to S_2}$

$V \in R_1 \to R_2$	Assumption
$V \in S_1 \to S_2$	Subsumption rule

[The other cases for rules for functions which have a single sort on the right hand side are essentially the same.]

Case: $\dfrac{R \sqsubset A_1 \to A_2 \qquad \Theta \vdash R \trianglelefteq U}{\Theta \vdash R \trianglelefteq U \sqcup S}$

$V \in R$	Assumed
$V \in S'$ for some S' in U	Ind. Hyp. (Same V, subderiv.)

Case: $\dfrac{R \sqsubset A_1 \to A_2 \quad \Theta \vdash R \trianglelefteq S}{\Theta \vdash R \trianglelefteq U \sqcup S}\,(U \neq \cdot)$

$V \in R$	Assumed
$V \in S$	Subsumption rule

□

Next we prove a similar result for inclusions $Z \preceq Y$, which has a useful corollary for the case when Y has no components.

Lemma 6.6.8 (Subsumption for sort unions)
If $V \in Z$ and $Z \preceq Y$ then $\vdash V \in S$ for some S in Y.

Proof: By induction on the structure of the derivation of $Z \preceq U$.

Case: $\dfrac{\cdot \vdash_\Sigma R \trianglelefteq \cdot}{R \preceq \bot}$

$V \in R$	Assumed
$V \in S$ with S in $\cdot$	Gen. Subs. Lem. (6.6.7)

Impossible, case cannot occur.

Case: $\dfrac{\cdot \vdash_\Sigma R \trianglelefteq S}{R \preceq S}$

By the subsumption rule.

Case: $\dfrac{Z_1 \preceq Y_1 \qquad Z_2 \preceq Y_2}{Z_1 \sqcup Z_2 \preceq \sqcup\{R \,|\, R \text{ in } Y_1 \text{ or } Y_2\}}$

Then, $V \in Z_1$ or $V \in Z_2$ (inversion)

So, $V \in S$ for some S in Y_1 or Y_2 (ind. hyp.)

Case: $\dfrac{}{\bot \preceq \bot}$

Then, $V \in \bot$ is impossible. (No rule matches this conclusion.)

Case cannot occur.

Case: $\dfrac{}{() \preceq 1}$

Then, $V = ()$ and $\vdash () \in 1$.

Case: $\dfrac{Z_1 \preceq Y_1 \qquad Z_2 \preceq Y_2}{(Z_1, Z_2) \preceq \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}}$

$V \in (Z_1, Z_2)$	Assumed
$V = (V_1, V_2)$ and $V_1 \in Z_1$ and $V_2 \in Z_2$	Inversion
$V_1 \in S_1$ for some S_1 in Y_1	Ind. Hyp.
$V_2 \in S_2$ for some S_2 in Y_2	Ind. Hyp.
$(V_1, V_2) \in S_1 \times S_2$	Rule for $\times$
$S_1 \times S_2$ in $\sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}$	

Case: $\dfrac{Z \preceq R_1 \sqcup \ldots \sqcup R_n \qquad c\,R_1 \rightarrowtail \rho_1 \;\; \ldots \;\; c\,R_n \rightarrowtail \rho_n}{c\,Z \preceq \rho_1 \sqcup \ldots \sqcup \rho_n}$

$V \in cZ$	Assumed
$V = cW$ and $W \in Z$	Inversion
$W \in R_i$ for some R_i in $R_1 \sqcup \ldots \sqcup R_n$	Ind. Hyp.
$cW \in \rho_i$	Rule for Constr. Appl.

□

Corollary 6.6.9 (Correctness of emptiness)
If $Z \preceq \bot$ then there is no V satisfying $V \in Z$.

Proof: Immediate from the above lemma. □

Next we prove a lemma which demonstrates that our definition of the inversion principle of a base refinement ρ is well behaved. We remark that this property fails if we instead directly use $\mathsf{body}(\rho)$ as the inversion principle for ρ.

Lemma 6.6.10 (Monotonicity of inversion principles)
If $\rho_1 \trianglelefteq \rho_2$ *and* $c\,S$ *in* $\mathsf{inv}(\rho_1)$
then $S \trianglelefteq R$ *and* $c\,R$ *in* $\mathsf{inv}(\rho_2)$ *for some* R.

Proof: By the definition of $\mathsf{inv}(\cdot)$ we have:

$$\mathsf{inv}(\rho_1) = \mathsf{body}(\&\{r \mid \rho_1 \trianglelefteq r\})$$
$$\mathsf{inv}(\rho_2) = \mathsf{body}(\&\{r \mid \rho_2 \trianglelefteq r\})$$

The result follows because $\rho_1 \trianglelefteq \rho_2$ implies:

$$\{r \mid \rho_2 \trianglelefteq r\} \subseteq \{r \mid \rho_1 \trianglelefteq r\}$$

$$\&\{r \mid \rho_1 \trianglelefteq r\} \trianglelefteq \&\{r \mid \rho_2 \trianglelefteq r\}$$

$$\mathsf{body}(\&\{r \mid \rho_1 \trianglelefteq r\}) \trianglelefteq \mathsf{body}(\&\{r \mid \rho_2 \trianglelefteq r\})$$

Applying the Generalized Subsumption Lemma (6.6.7) yields the required result.

□

Next we have a lemma that demonstrates that our inversion principles are consistent with the values inhabiting datasorts.

Lemma 6.6.11 (Strong value inversion for datasorts)
If $V \in \rho$ *then* $V = c\,W$ *for some* c *and* W
satisfying $W \in S$ *and* $c\,S$ *in* $\mathsf{inv}(\rho)$.

Proof:

$V = c\,W$ with $W \in R$ and $c\,R$ in $\mathsf{body}(\rho')$ and $\rho' \trianglelefteq \rho$	Weak Inv. Lemma (6.6.3)
$\mathsf{inv}(\rho) = \mathsf{body}(\&\{r \mid \rho \trianglelefteq r\})$	Def. $\mathsf{inv}(\cdot)$
$\rho \trianglelefteq \&\{r \mid \rho \trianglelefteq r\}$	& is g.l.b. (5.8.11)
$\rho' \trianglelefteq \&\{r \mid \rho \trianglelefteq r\}$	Transitivity (5.8.11)
$\mathsf{body}(\rho') \trianglelefteq \mathsf{inv}(\rho)$	Lemma 5.6.2
$W \in S$ with $c\,S$ in $\mathsf{inv}(\rho)$	Gen. Subs. Lemma (6.6.7)

□

Lemma 6.6.12 (Value inversion for pairs)
If $V \in R_1 \times R_2$ *then* $V = (V_1, V_2)$ *with* $V_1 \in R_1$ *and* $V_2 \in R_2$.

Proof:

$V : A_1 \times A_2$ for some A_1 and A_2	Well-formedness of $V \in S_1 \times S_2$
$V = (V_1, V_2)$	Inversion
$V_1 \in S_1$ and $V_2 \in S_2$ with $S_1 \times S_2 \trianglelefteq R_1 \times R_2$	Lemma 6.6.4
($S_1 \trianglelefteq R_1$ and $S_2 \trianglelefteq R_2$) or $S_1 \trianglelefteq \cdot$ or $S_2 \trianglelefteq \cdot$	Prod. Inv. Lemma 6.6.6
If $S_1 \trianglelefteq \cdot$ then:	
$V_1 \in S_1$ with S_1 in $\cdot$	Gen. Subs. Lemma (6.6.7)
Contradiction.	
(The case for $S_2 \trianglelefteq \cdot$ is dual.)	
So, $S_1 \trianglelefteq R_1$ and $S_2 \trianglelefteq R_2$	
$V_1 \in R_1$ and $V_2 \in R_2$	Subsumption Rule

□

We continue by proving a substitution lemma for the situation where a substitution σ is applied to a sort assignment derivation with a corresponding pattern sort assumption.

Lemma 6.6.13 (Pattern substitution)
If $V \in Z$ *and* $V = \sigma P$ *and* $P \in Z \vdash M \in R$
then $\sigma M \in R$.

Proof: By induction on the derivation of $P \in Z \vdash M \in R$, generalizing to:
If $V_1 \in Z_1$ and ... and $V_n \in Z_n$, and $V_1 = \sigma P_1$ and ... and $V_n = \sigma P_n$,
and $\Delta_2 = P_1 \in Z_1, \ldots, P_n \in Z_n$
and the domain of σ excludes variables bound in Δ_1 then:

1. If $\Delta_1, \Delta_2 \vdash M \in R$ then $\Delta_1 \vdash \sigma M \in R$.
2. If $\Delta_1, \Delta_2 \vdash S \bowtie \Omega \in R$ then $\Delta_1 \vdash S \bowtie \sigma\Omega \in R$.

Case: $\dfrac{x \in R \text{ in } \Delta_2}{\Delta_1, \Delta_2, \vdash x \in R}$

$P_i = x$ and $V_i \in R$ and $V_i = \sigma\, x$	Assumed
$\vdash \sigma\, x \in R$	Since $V_i = \sigma\, x$

Case: $\dfrac{x \in R \text{ in } \Delta_1}{\Delta_1, \Delta_2 \vdash x \in R}$

$\sigma\, x = x$	σ excludes Δ_1
$\Delta_1 \vdash x \in R$	Variable rule

Case: $\dfrac{u{\in}R \text{ in } \Delta_1}{\Delta_1, \Delta_2 \vdash u \in R}$

As for the previous case.

Case: $\dfrac{\Delta_1, x{\in}R, \Delta_2 \vdash M \in S}{\Delta_1, \Delta_2 \vdash \lambda x{:}A\,.M \in R \to S}$

Then, rename x so that x is not in σ

$\Delta_1, x{\in}R \vdash \sigma\, M \in S$	Ind. Hyp.
$\Delta_1 \vdash \lambda x{:}A\,.(\sigma\, M) \in R \to S$	Rule for λ
$\Delta_1 \vdash \sigma\,(\lambda x{:}A\,.M) \in R \to S$	x not in σ

Case: $\dfrac{\Delta_1, \Delta_2 \vdash M \in R \qquad \Delta_1, \Delta_2 \vdash R \bowtie \Omega \in S}{\Delta_1, \Delta_2 \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S}$

$\Delta_1 \vdash \sigma M \in R$	Ind. Hyp.
$\Delta_1 \vdash R \bowtie \sigma\Omega \in S$	Ind. Hyp.
$\Delta_1 \vdash \sigma(\mathbf{case}\ M\ \mathbf{of}\ \Omega \in S)$	Rule for **case**

Case: [The remaining right rules.]

Each similarly rebuilds the derivation, after applying the induction hypothesis.

Case: $\dfrac{\Delta_1, P{\in}Z, \Delta_2 \vdash M{\in}R \qquad \Delta_1, \Delta_2 \vdash (Z \backslash P) \bowtie \Omega \in R}{\Delta_1, \Delta_2 \vdash Z \bowtie (P \Rightarrow M \mid \Omega) \in R}$

Then, rename var's bound by P in $P \Rightarrow M$ so that σ excludes P

$\Delta_1, P{\in}Z \vdash \sigma M \in R$	Ind. Hyp.
$\Delta_1 \vdash (Z \backslash P) \bowtie \sigma\Omega \in R$	Ind. Hyp.
$\Delta_1 \vdash Z \bowtie (P \Rightarrow \sigma M \mid \sigma\Omega) \in R$	Rule
$\Delta_1 \vdash Z \bowtie \sigma(P \Rightarrow M \mid \Omega) \in R$	

Case: $\dfrac{Z \preceq \bot}{\Delta_1, \Delta_2 \vdash Z \bowtie \cdot \in R}$

$\Delta_1 \vdash Z \bowtie \cdot \in R$	Rule

Case: $\dfrac{(\Delta_1', P_1'{\in}Z_1', P_2'{\in}Z_2'), \Delta_2 \vdash M{\in}R}{(\Delta_1', (P_1', P_2') \in (Z_1', Z_2')), \Delta_2 \vdash M{\in}R}$

σ excludes var's in $\Delta_1', (P_1', P_2') \in (Z_1', Z_2') \quad (= \Delta_1)$	Assumed
σ excludes var's in $\Delta_1', P_1'{\in}Z_1', P_2'{\in}Z_2'$	
$\Delta_1', P_1'{\in}Z_1', P_2'{\in}Z_2' \vdash \sigma M{\in}R$	Ind. Hyp.
$\Delta_1', (P_1', P_2') \in (Z_1', Z_2') \vdash \sigma M{\in}R$	Left rule for pairs

Case: $\dfrac{(\Delta_1', P{\in}Z_1'), \Delta_2 \vdash M{\in}R \qquad (\Delta_1', P{\in}Z_2'), \Delta_2 \vdash M{\in}R}{(\Delta_1', P \in Z_1' \sqcup Z_2'), \Delta_2 \vdash M{\in}R}$

σ excludes var's in $\Delta_1', P \in Z_1' \sqcup Z_2' \quad (= \Delta_1)$ — Assumed

σ excludes var's in $\Delta_1', P \in Z_1'$ and $\Delta_1', P \in Z_2'$

$\Delta_1', P{\in}Z_1' \vdash \sigma M{\in}R$ — Ind. Hyp.

$\Delta_1', P{\in}Z_2' \vdash \sigma M{\in}R$ — Ind. Hyp.

$\Delta_1', P \in Z_1' \sqcup Z_2' \vdash \sigma M{\in}R$ — Left rule for $\sqcup$

Case: [The remaining left rules involving Δ_1.]

Each similarly rebuilds the derivation, after applying the induction hypothesis.

Case: $\dfrac{\Delta_1, (\Delta_2', P_1'{\in}Z_1', P_2'{\in}Z_2') \vdash M{\in}R}{\Delta_1, (\Delta_2', (P_1', P_2') \in (Z_1', Z_2')) \vdash M{\in}R}$

$V_i \in (Z_1', Z_2')$ — Assumed

$V_i = (W_1, W_2)$ with $W_1{\in}Z_1'$ and $W_2{\in}Z_2'$ — Inversion

$(W_1, W_2) = \sigma(P_1', P_2')$ — Assumed

$W_1 = \sigma P_1'$ and $W_2 = \sigma P_2'$

$\Delta_1 \vdash \sigma M{\in}R$ — Ind. Hyp.

Case: $\dfrac{\Delta_1, (\Delta_2', P_1'{\in}S_1, P_2'{\in}S_2) \vdash M{\in}R}{\Delta_1, (\Delta_2', (P_1', P_2') \in S_1{\times}S_2) \vdash M{\in}R}$

$V_i \in S_1 \times S_2$ — Assumed

$V_i = (W_1, W_2)$ with $W_1{\in}S_1$ and $W_2{\in}S_2$ — Value Inv. Lem. (6.6.12)

$V_i = \sigma(P_1', P_2')$ — Assumed

$W_1 = \sigma P_1'$ and $W_2 = \sigma P_2'$

$\Delta_1 \vdash \sigma M{\in}R$ — Ind. Hyp.

Case: $\dfrac{\Delta_1, \Delta_2' \vdash M{\in}R}{\Delta_1, (\Delta_2', () \in ()) \vdash M{\in}R}$

$\Delta_1 \vdash \sigma M{\in}R$ — Ind. Hyp.

Case: $\dfrac{\Delta_1, \Delta_2' \vdash M{\in}R}{\Delta_1, (\Delta_2', () \in 1) \vdash M{\in}R}$

$\Delta_1 \vdash \sigma M{\in}R$ — Ind. Hyp.

Case: $\dfrac{\Delta_1, (\Delta_2', P{\in}Z) \vdash M{\in}R}{\Delta_1, (\Delta_2',\, c\,P {\in} c\,Z) \vdash M{\in}R}$

$V_i{\in}cZ$	Assumed
$V_i = cW$ with $W{\in}Z$	Inversion
$V_i = \sigma(cP)$	Assumed
$W = \sigma P$	
$\Delta_1 \vdash \sigma M{\in}R$	Ind. Hyp.

Case: $\dfrac{(c \neq c_2)}{\Delta_1, (\Delta_2',\, c\,P {\in} c_2\,Z) \vdash M{\in}R}$

$V_i{\in}c_2\,Z$	Assumed
$V_i = c_2 W$ with $W{\in}Z$	Inversion
$V_i = \sigma(cP) = c(\sigma P)$	Assumed
Contradiction, case cannot occur.	

Case: $\dfrac{\Delta_1, (\Delta_2',\, c\,P {\in} \mathsf{inv}(\rho)) \vdash M{\in}R}{\Delta_1, (\Delta_2',\, c\,P {\in} \rho) \vdash M{\in}R}$

$V_i{\in}\rho$	Assumed
$V_i = cW$ with $W{\in}S$ and cS in $\mathsf{inv}(\rho)$	Strong Inv. Lem. (6.6.11)
$cW \in cS$	Rule for cZ
$cW \in \mathsf{inv}(\rho)$	Rule for $\sqcup$, repeatedly
$\Delta_1 \vdash \sigma M{\in}R$	Ind. Hyp.

Case: $\dfrac{Z_i \preceq Y \qquad \Delta_1, (\Delta_2', x{\in}Y) \vdash M{\in}R}{\Delta_1, (\Delta_2', x{\in}Z_i) \vdash M{\in}R}$

$V_i{\in}Z_i$	Assumed
$V_i{\in}S$ with S in Y	Sort Union Subs. Lem. (6.6.8)
$V_i{\in}Y$	Rule for $\sqcup$, repeatedly
$\Delta_1 \vdash \sigma M{\in}R$	Ind. Hyp.

Case: $\dfrac{S \preceq Y \qquad \Delta_1, (\Delta_2', P_i{\in}Y) \vdash M{\in}R}{\Delta_1, (\Delta_2', P_i{\in}S) \vdash M{\in}R}$

$V_i{\in}S$	Assumed
$V_i{\in}S'$ with S' in Y	Sort Union Subs. Lem. (6.6.8)
$V_i{\in}Y$	Rule for $\sqcup$, repeatedly
$\Delta_1 \vdash \sigma M{\in}R$	Ind. Hyp.

Case: $\dfrac{\Delta_1, (\Delta_2', P_i \in Z_1') \vdash M \in R \qquad \Delta_1, (\Delta_2', P_i \in Z_2') \vdash M \in R}{\Delta_1, (\Delta_2', P_i \in Z_1' \sqcup Z_2') \vdash M \in R}$

$V_i \in Z_1' \sqcup Z_2'$	Assumed
$V_i \in Z_1'$ or $V_i \in Z_2'$	Inversion
$\Delta_1 \vdash \sigma M \in R$	Ind. Hyp. (in each case)

Case: $\dfrac{}{\Delta_1, (\Delta_2', P_i \in \bot) \vdash M \in R}$

$V_i \in \bot$	Assumed
Impossible, case cannot occur. (No rule matches this conclusion.)	□

Next we prove a lemma that justifies our use of pattern subtraction to produce a pattern sort characterizing values not matched by a pattern. This is required in the proof of progress for the case where we have an expression of the form $\mathbf{case}\, V\, \mathbf{of}\, P \Rightarrow M \mid \Omega$.

Lemma 6.6.14 (Coverage of pattern subtraction)
If $V \in Z$ and $\Gamma \vdash P : A$ and $Z \sqsubset A$ then one of the following holds:

1. *$V = \sigma P$ for some σ containing only variables in P*
2. *$V \in Z \backslash P$*

Proof: By induction on P, Z lexicographically, following the definition of $Z \backslash P$.

Case: $P \backslash x = \bot$

Then $V = \sigma x$, choosing $\sigma = \{V/x\}$.

Case: $(Z_1, Z_2) \setminus (P_1, P_2) = ((Z_1 \backslash P_1), Z_2) \sqcup (Z_1, (Z_2 \backslash P_2))$

$V \in (Z_1, Z_2)$	Assumed
$V = (V_1, V_2)$ with $V_1 \in Z_1$ and $V_2 \in Z_2$	Inversion
$V_1 = \sigma_1 P_1$ or $V_1\in Z_1 \backslash P_1$	Ind. Hyp.
$V_2 = \sigma_2 P_2$ or $V_2 \in Z_2 \backslash P_2$	Ind. Hyp.
$(V_1, V_2) = (\sigma_1 P_1, \sigma_2 P_2)$ or $(V_1, V_2) \in ((Z_1 \backslash P_1), Z_2)$ or $(V_1, V_2) \in (Z_1, (Z_2 \backslash P_2))$	
$(V_1, V_2) = (\sigma_1 \sigma_2)(P_1, P_2)$	σ_1, σ_2 disjoint
or $(V_1, V_2) \in ((Z_1 \backslash P_1), Z_2) \sqcup (Z_1, (Z_2 \backslash P_2))$	Rule for $\sqcup$

Case: $R \setminus (P_1, P_2) = ((R_1 \backslash P_1), R_2) \sqcup (R_1, (R_2 \backslash P_2))$ with $R \overset{\rightrightarrows}{\simeq} R_1 \otimes R_2$

$V \in R$	Assumed
$R \trianglelefteq R_1 \times R_2$	Lemma 5.6.8
$V \in R_1 \times R_2$	Subsumption
$V = (V_1, V_2)$ with $V_1 \in R_1$ and $V_2 \in R_2$	Val. Inv. Lem. 6.6.12

Then, following the previous case.

Case: $Z \setminus () \;=\; \bot$

$V \in Z$	Assumed
$\Gamma \vdash () : A$	Assumed
$A = 1$	Inversion
$Z \sqsubset 1$	Assumed
$V : 1$	Well-formedness of $V \in Z$
$V = ()$	Inversion
$V = \sigma()$ where $\sigma = \cdot$ (empty substitution)	

Case: $(c\,Z) \setminus (c\,P) \;=\; c\,(Z \setminus P)$

$V \in cZ$	Assumed
$V = cW$ with $W \in Z$	Inversion
$W = \sigma P$ or $W \in Z \setminus P$	Ind. Hyp.
$cW = \sigma(cP)$	Def. Substitution
or $cW \in c(Z \setminus P)$	Rule for cW

Case: $(c\,Z) \setminus (c'\,P) \;=\; c\,Z$ with $c \neq c'$

$V \in cZ$	Assumed

Case: $\rho \setminus (c\,P) \;=\; \bigsqcup\{c(R \setminus P) \mid c\,R \text{ in } \mathsf{inv}(\rho)\} \;\sqcup\; \bigsqcup\{c'\,R' \mid c'\,R' \text{ in } \mathsf{inv}(\rho) \text{ and } c' \neq c\}$

$V \in \rho$	Assumed
$V = c'W$ with $W \in S$ and $c'\,S$ in $\mathsf{inv}(\rho)$	Strong Value Inv. (6.6.11)
Suppose $c' = c$:	
$c(S \setminus P)$ in $\rho \setminus (cP)$	
$W = \sigma P$ or $W \in S \setminus P$	Ind. Hyp.
$cW = \sigma(cP)$ or $cW \in c(S \setminus P)$	Def. Subst., Rule for cZ
$cW = \sigma(cP)$ or $cW \in \rho \setminus (cP)$	Rule for $\sqcup$ repeatedly
Suppose $c' \neq c$:	
$c'\,S$ in $\rho \setminus (cP)$	
$c'W \in c'\,S$	
$c'W \in \rho \setminus (cP)$	Rule for $\sqcup$ repeatedly

Case: $\bot \setminus P \;=\; \bot$

$V \in \bot$	Assumed
Impossible, case cannot occur	

Case: $(Z_1 \sqcup Z_2) \setminus P \;=\; (Z_1 \backslash P) \sqcup (Z_2 \backslash P)$

$V \in Z_1 \sqcup Z_2$	Assumed
$V \in Z_1$ or $V \in Z_2$	Inversion
Assuming the former (the latter is dual):	
$V = \sigma P$ or $V \in Z_1 \backslash P$	Ind. Hyp.
$V = \sigma P$ or $V \in (Z_1 \backslash P) \sqcup (Z_2 \backslash P)$	Rule for $\sqcup$

□

The statement of our progress and sort preservation theorem requires a very slightly different notion of sort assignment than provided by our declarative rules. This is because sort preservation can be temporarily violated: when we perform a reduction of the form

$$E[\mathbf{case}\, V\, \mathbf{of}\; P \Rightarrow M \mid \Omega] \;\mapsto\; E[\mathbf{case}\, V\, \mathbf{of}\; \Omega]$$

and V is assigned the sort R in the given sorting derivation for the former expression, we find ourselves wanting to assign V a pattern sort $R \backslash P$ in the latter expression. But, the sort assignment rules do not allow us to assign pattern sorts to terms, only sorts (recall that the rules for assigning pattern sorts to values are part of a separate judgment that was only introduced to allow the technical development in this section).

This situation might seem awkward, but this problem can be easily fixed, in one of three ways.

1. We could alter the operational semantics so that the appropriate branch is always chosen in a single step, thus avoiding the need to assign pattern sorts to the intermediate steps.

2. Similarly, we could alter the statement of the theorem so that it only asserts the existence of a sequence of reductions $M \mapsto^+ N$ of length at least one, resulting in a term with the same sort as the input.

3. Alternatively, we could add the rules assigning pattern sorts to values to the sort assignment judgment.

The first option seems less than ideal because it complicates the semantics of pattern matching somewhat. The second option results in a somewhat less precise theorem than the one that we prove: in particular it does not indicate the situations where sort preservation is temporarily violated. The third option seems excessive, since the rules added would not be required for any programs, only for states during the execution of programs.

We thus make the following definition, which allows us to prove a theorem that seems to most accurately reflects the situation, and that can easily be modified to obtain proofs of any of the three alternative theorems mentioned above.

Definition 6.6.15 *$M \mathrel{\widehat{\in}} R$ means that one of the following holds.*

1. $M \in R$
2. $M = E[\mathbf{case}\, V\, \mathbf{of}\, \Omega]$ *with* $V \in Z$ *and* $\vdash Z \bowtie \Omega \in S$ *and* $u \in S \vdash E[u] \in R$.

The following lemma allows us to compose derivations involving $N \mathrel{\widehat{\in}} S$ without requiring an explicit case analysis each time.

Lemma 6.6.16 (Expression Substitution for $\widehat{\in}$)
If $u{\in}R_1 \vdash E[u] \in R_2$ *and* $M \mathrel{\widehat{\in}} R_1$ *then* $E[M] \mathrel{\widehat{\in}} R_2$.

Proof:
If $M{\in}R_1$ then $E[M] \in R_2$, by the Expression Substitution Lemma (6.6.2).
If $M = E'[\mathbf{case}\, V\, \mathbf{of}\, \Omega]$ with $V{\in}Z$ and $\vdash Z \bowtie \Omega \in S$ and $u{\in}S \vdash E'[u] \in R_1$
then:

$E[M] = E_2[\mathbf{case}\, V\, \mathbf{of}\, \Omega]$ where $E_2[\cdot] = E[E'[\cdot]]$

$u{\in}S \vdash E_2[u] \in R_2$ Expr. Subst. Lem.

$E[M] \mathrel{\widehat{\in}} R_2$ Def. $\widehat{\in}$

□

Lemma 6.6.17 (Case reduction)
If $V{\in}Z$ *and* $\vdash Z \bowtie \Omega \in S$ *and* $u{\in}S \vdash E[u] \in R$
then $E[\mathbf{case}\, V\, \mathbf{of}\, \Omega] \mapsto E[N]$ *with* $E[N] \mathrel{\widehat{\in}} R$.

Proof: We have the following cases for $\mathcal{D} :: Z \bowtie \Omega \in S$

Case: $\dfrac{Z \preceq \bot}{\vdash Z \bowtie \cdot \in S}$

$V{\in}Z$ Assumed

Impossible, case cannot occur Correctness of Emptiness (6.6.9)

Case: $\dfrac{P{\in}Z \vdash M'{\in}S \qquad \vdash (Z \backslash P) \bowtie \Omega' \in S}{\vdash Z \bowtie (P \Rightarrow M' \mid \Omega') \in S}$

Suppose there exists σ such that $V = \sigma P$:

$E[\mathbf{case}\, V\, \mathbf{of}\, (P \Rightarrow M' \mid \Omega')] \mapsto E[\sigma M']$ Red. Rule

$V{\in}Z$ Assumed

$\sigma M' \in S$ Pat. Subst. Lem. (6.6.13)

$E[\sigma M'] \in R$ Expr. Subst. Lem. (6.6.2)

Otherwise, for all σ, $V \neq \sigma P$, and so:

$E[\mathbf{case}\, V\, \mathbf{of}\, (P \Rightarrow M' \mid \Omega')] \mapsto E[\mathbf{case}\, V\, \mathbf{of}\, \Omega']$ Red. Rule

$V \in Z \backslash P$ Coverage of Pat. Subtr. (6.6.14)

$\vdash (Z \backslash P) \bowtie \Omega' \in S$ Assumed (premise of rule)

$u{\in}S \vdash E[u] \in R$ Assumed (Def. $\widehat{\in}$ above)

$E[\mathbf{case}\, V\, \mathbf{of}\, \Omega'] \mathrel{\widehat{\in}} R$ Def. $\widehat{\in}$

□

Lemma 6.6.18 (Value inversion for $\to$)
If $V \in R$ and $R \trianglelefteq S_1 \to S_2$
then $V = \lambda x.M$ with $x \in S_1 \vdash M \in S_2$.

Proof: By induction on the structure of $\mathcal{D} :: V \in R$.

The cases for variables cannot occur, since the context is empty.

The cases for applications, **fix**, **case** are not values.

The cases for $c\,M$, (M_1, M_2) and $()$ cannot occur, since they lead to sorts incompatible with $S_1 \to S_2$.

Thus, the following are the only possible cases.

Case: $\dfrac{x \in R_1 \vdash M \in R_2}{\vdash \lambda x{:}A\,.M \in R_1 \to R_2}$

$R_1 \to R_2 \trianglelefteq S_1 \to S_2$	Assumed
$S_1 \trianglelefteq R_1$ and $R_2 \trianglelefteq S_2$	Inversion
$x \in S_1 \vdash x \in R_1$	Rule for x, Subsumption
$x \in S_1 \vdash \{x/x\}M \in R_2$	Pattern Subst. Lem. (6.6.13)
$x \in S_1 \vdash M \in S_2$	Subsumption Rule

Case: $\dfrac{\vdash V \in R_2 \qquad R_2 \trianglelefteq R}{\vdash V \in R}$

$R \trianglelefteq S_1 \to S_2$	Assumed
$R_2 \trianglelefteq S_1 \to S_2$	Transitivity (5.8.11)
$V = \lambda x.M$ with $x \in S_1 \vdash M \in S_2$	Ind. Hyp.

Case: $\dfrac{\vdash V \in R_1 \qquad \vdash V \in R_2}{\vdash V \in R_1 \,\&\, R_2}$

$R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2$	Assumed
$R_1 \trianglelefteq S_1 \to S_2 \quad$ or $\quad R_2 \trianglelefteq S_1 \to S_2$	Inversion
$V = \lambda x.M$ with $x \in S_1 \vdash M \in S_2$	Ind. Hyp. (in each case)

Case: $\dfrac{}{\vdash V \in \top^A}$

$\top^A \trianglelefteq S_1 \to S_2$	Assumed

Impossible, case cannot occur.

□

Lemma 6.6.19 (Contextual Evaluation)
If $M \mapsto N$ then $E'[M] \mapsto E'[N]$.

Proof: By analysis of cases for $M \mapsto N$.

We recall that we have the following rules for $\mapsto$.

$$\begin{array}{rcll} E[(\lambda x.M)\,V] & \mapsto & E[\{V/x\}M] & \\ E[\textbf{case}\ V\ \textbf{of}\ P \Rightarrow M \mid \Omega] & \mapsto & E[\sigma M] & (\text{if } V = \sigma P) \\ E[\textbf{case}\ V\ \textbf{of}\ P \Rightarrow M \mid \Omega] & \mapsto & E[\textbf{case}\ V\ \textbf{of}\ \Omega] & (\nexists \sigma.V = \sigma P) \\ E[\textbf{case}\ V\ \textbf{of}\ \cdot] & \mapsto & \mathit{MatchException} & \\ E[\textbf{fix}\ u.M] & \mapsto & E[\{\textbf{fix}\ u.M/u\}M] & \end{array}$$

Then, for the four rules of the form $E[M'] \mapsto E[N']$, the result holds, since also $E'[E[M']] \mapsto E'[E[N']]$.

Further, the rule involving *MatchException* does not match our assumption that $M \mapsto N$, hence is outside the scope of the lemma. □

Finally, we have the main theorem of this section.

Theorem 6.6.20 (Progress and sort preservation)
If $M \mathrel{\widehat{\in}} R$ then one of the following holds.

1. *M is a value.*
2. *$M \mapsto N$, with $N \mathrel{\widehat{\in}} R$.*

Proof:
When $M{\in}R$: by induction on the structure of the derivation of $M{\in}R$. The cases appear below.

Otherwise: the second part of the definition of $\widehat{\in}$ matches the Case Reduction Lemma (6.6.17), which yields the required result.

The cases for the induction when $M{\in}R$ are as follows.

Case: $\dfrac{\vdash M' \in S \qquad \vdash S \bowtie \Omega \in R}{\vdash \textbf{case}\ M'\ \textbf{of}\ \Omega \in R}$

$M' = V$ or $M' \mapsto N$ with $N \mathrel{\widehat{\in}} S$ — Ind. Hyp.

If $M' = V$ then:

$u{\in}R \vdash E[u] \in R$ where $E = [\,]$ — Var. Rule

case M' **of** $\Omega \mapsto N$ with $N \mathrel{\widehat{\in}} R$ — Case Red. Lem. (6.6.17)

Otherwise $M' \mapsto N$ with $N \mathrel{\widehat{\in}} S$. Then:

case M' **of** $\Omega \mapsto$ **case** N **of** Ω — Context Eval. Lem. (6.6.19)

$u{\in}S \vdash$ **case** u **of** $\Omega \in R$ — Rule for **case**

$\vdash$ **case** N **of** $\Omega \mathrel{\widehat{\in}} R$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Case: $\dfrac{x{\in}R_1 \vdash M' \in R_2}{\vdash \lambda x.M' \in R_1 \to R_2}$

$\lambda x.M'$ is a value.

Case: $\dfrac{\vdash M_1 \in R_2 \to R \qquad \vdash M_2 \in R_2}{\vdash M_1\, M_2 \in R}$

$M_1 = V_1$ or $M_1 \mapsto N_1$ with $N_1 \widehat{\in} R_2 \to R$ — Ind. Hyp.

If $M_1 \mapsto N_1$ with $N_1 \widehat{\in} R_2 \to R$:

$M_1\, M_2 \mapsto N_1\, M_2$ — Context Eval. Lem. (6.6.19)
$u{\in}R_2 \to R \vdash u\, M_2 \in R$ — Rules for u, Appl.
$\vdash N_1\, M_2 \mathrel{\widehat{\in}} R$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Otherwise $M_1 = V_1$, and then:

$V_1 = \lambda x.M_1'$ and $x{\in}R_2 \vdash M_1' \in R$ — Value Inv. for $\to$ (6.6.18)
$M_2 = V_2$ or $M_2 \mapsto N_2$ with $N_2 \widehat{\in} R_2$ — Ind. Hyp.

If $M_2 \mapsto N_2$ with $N_2 \widehat{\in} R_2$:

$V_1\, M_2 \mapsto V_1\, N_2$ — Context Eval. Lem. (6.6.19)
$u{\in}R_2 \vdash V_1\, u \in R$ — Rules for u, Appl.
$\vdash V_1\, N_2 \mathrel{\widehat{\in}} R$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Otherwise $M_2 = V_2$, and then:

$(\lambda x.M_1')\, V_2 \mapsto \{V_2/x\}M_1'$ — Red. Rule
$\{V_2/x\}M_1' \in R$ — Pat. Subst. Lem. (6.6.13)

Case: $\dfrac{u{:}R \vdash M' \in R}{\vdash \mathbf{fix}\, u.M' \in R}$

$\mathbf{fix}\, u.M' \mapsto \{\mathbf{fix}\, u.M'/u\}M'$ — Red. Rule
$\{\mathbf{fix}\, u.M'/u\}M' \in R$ — Expr. Subst. Lem. (6.6.2)

Case: $\dfrac{c\, S \rightarrowtail \rho \qquad \vdash M' \in S}{\vdash c\, M' \in \rho}$

$M' = V'$ or $M' \mapsto N'$ with $N' \widehat{\in} S$ — Ind. Hyp.

If $M' \mapsto N'$ with $N' \widehat{\in} S$:

$c\, M' \mapsto c\, N'$ — Context Eval. Lem. (6.6.19)
$u{\in}S \vdash c\, u \in \rho$ — Rules for u, Constr. Appl.
$\vdash c\, N' \mathrel{\widehat{\in}} \rho$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Otherwise $M_1 = V_1$, and then:

$c\, V_1$ is a value.

Case: $\dfrac{\vdash M_1 \in R_1 \qquad \vdash M_2 \in R_2}{\vdash (M_1, M_2) \in R_1 \times R_2}$

$M_1 = V_1$ or $M_1 \mapsto N_1$ with $N_1 \widehat{\in} R_1$ — Ind. Hyp.

If $M_1 \mapsto N_1$ with $N_1 \widehat{\in} R_1$:

$(M_1,M_2) \mapsto (N_1,M_2)$ — Context Eval. Lem. (6.6.19)

$u \in R_2 \to R \vdash (u,M_2) \in R$ — Rules for u, Appl.

$\vdash (N_1,M_2) \mathrel{\widehat{\in}} R$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Otherwise $M_1 = V_1$, and then:

$M_2 = V_2$ or $M_2 \mapsto N_2$ with $N_2 \widehat{\in} R_2$ — Ind. Hyp.

If $M_2 \mapsto N_2$ with $N_2 \widehat{\in} R_2$:

$(V_1,M_2) \mapsto (V_1,N_2)$ — Context Eval. Lem. (6.6.19)

$u \in R_2 \vdash (V_1,u) \in R$ — Rules for u, Appl.

$\vdash (V_1,N_2) \mathrel{\widehat{\in}} R$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Otherwise $M_2 = V_2$, and then:

(V_1,V_2) is a value.

Case: $\dfrac{}{\vdash () \in 1}$

() is a value.

Case: $\dfrac{\vdash M \in S \qquad S \trianglelefteq R}{\vdash M \in R}$

$M = V$ or $M \mapsto N$ with $N \widehat{\in} S$ — Ind. Hyp.

When $M = V$: as required.

When $M \mapsto N$ with $N \widehat{\in} S$:

$u \in S \vdash u \in R$ — Rules for u, Subsumption

$\vdash N \mathrel{\widehat{\in}} R$ — Expr. Subst. for $\widehat{\in}$ (6.6.16)

Case: $\dfrac{\vdash V \in R \qquad \vdash V \in S}{\vdash V \in R \,\&\, S}$

Immediate.

Case: $\dfrac{}{\vdash V \in \top^A}$

Immediate.

Case: Left rules and variable rules: cannot occur since the context is empty. □

6.7 Bidirectional sort checking with pattern matching

In this section we show how to extend bidirectional sort checking the language of this chapter, and in particular to the sequential pattern matching construct.

6.7.1 Syntax

We have the following syntax for inferable and checkable terms, following Section 3.11. Branches are designated as always containing checkable terms. We require the body of a recursive expression $\mathbf{fix}u.M$ to be an annotated expression $(C{\in}L)$ in order to force the sorts assigned to u to be explicit; the resulting expression is inferable. We include pairs of checkable terms as checkable, and an empty product as inferable, but do not include pairs of inferable terms as inferable. The latter seems less essential, and does not fit with our general principle that introduction forms are checkable, while elimination forms are inferable. Further, it complicates the annotatability proof. (However, our implementation does allow inferable pairs.)

We also introduce a restriction Π of contexts for those containing only ordinary assumptions that do not involve patterns, since our algorithm transforms pattern sort assumptions into such assumptions prior to checking the body of a branch.

$$
\begin{array}{rrcl}
\text{Inferable Terms} & I & ::= & x \mid u \mid c\,C \mid I\,C \mid () \mid \mathbf{fix}\,u{:}A.(C{\in}L) \mid (C{\in}L) \\
\text{Checkable Terms} & C & ::= & I \mid \lambda x{:}A.C \mid (C_1, C_2) \mid \mathbf{case}\ I\ \mathbf{of}\ \Omega^{\mathsf{c}} \\
\text{Checkable Branches} & \Omega^{\mathsf{c}} & ::= & \cdot \mid (P \Rightarrow C \mid \Omega^{\mathsf{c}}) \\
\text{Non-pattern context} & \Pi & ::= & \cdot \mid x{\in}R \mid u{\in}R
\end{array}
$$

6.7.2 Sort checking algorithm

The judgments for inferring and checking sorts for term constructs are based on those in Section 3.11, with additional judgments based on Section 6.5 for checkable branches, pattern contexts with a checkable branch body, and an algorithmic form of the inclusion $Z \preceq Y$ that synthesizes Y. We additionally have a judgment $R \overset{\Rightarrow}{\trianglelefteq} S$ which non-deterministically extracts the conjuncts of refinements of function types. Finally, we have a judgment $A \overset{\Rightarrow}{\perp} R$ that synthesizes empty refinements of a given type, when they exist.

$\Pi \vdash_\Sigma I \overset{\Rightarrow}{\in} R$	Term I has R as an inferable sort under context Π. (Where $\Gamma \vdash I : A$ and $\Pi \sqsubset \Gamma$ and $R \sqsubset A$.)
$\Pi \vdash_\Sigma C \overset{\Leftarrow}{\in} R$	Term C checks against sort R under context Π. (Where $\Gamma \vdash C : A$ and $\Pi \sqsubset \Gamma$ and $R \sqsubset A$.)
$\Pi \vdash_\Sigma Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$	Matching pattern sort Z with branches Ω^{c} checks against sort R under context Π. (Where $\Gamma \vdash_\Sigma S \bowtie \Omega^{\mathsf{c}} : A$ and $\Pi \sqsubset \Gamma$ and $Z \sqsubdot S$ and $R \sqsubset A$.)
$\Pi; \Delta \vdash_\Sigma C \overset{\Leftarrow}{\in} R$	Term C checks against sort R under context Π and pattern context Δ. (Where $\Gamma_1; (\Phi_1, \Phi_2) \vdash C : A$ and $\Pi \sqsubset \Gamma_1; \Phi_2$ and $\Delta \sqsubset \cdot; \Phi_2$ and $\Phi_2 \cong \Gamma_2$ and $R \sqsubset A$.)
$R \overset{\Rightarrow}{\trianglelefteq} S$	S is synthesized as a conjunct of function sort R (Where $R \sqsubset A \to B$ and $S \sqsubset A \to B$.)
$\vdash_\Sigma Z \overset{\Rightarrow}{\preceq} Y$	Y is a minimum sort union covering pattern sort Z. (Where $Z \sqsubdot A$ and $Y \sqsubdot A$.)
$\vdash_\Sigma A \overset{\Rightarrow}{\top} R$	R is synthesized as an empty refinement of A. (Where $R \sqsubset A$.)

The rules for these judgments are based directly on the rules for term constructs from Section 3.11, and the rules for branches and left rules from Section 6.5 but with a checkable term on the right. We add checking rules for products, recursion, and **case**, and use a more algorithmic form of the constructor application rule.

In the left rules, we restrict the use of inclusions $Z \overset{\Rightarrow}{\preceq} Y$ to variable patterns via a rule that also performs the case analysis designated by the sort union Y, and thus creates ordinary, non-pattern assumptions in the context Π. When all assumptions have been converted to non-pattern assumptions, we have a rule that commences checking the term constructs in the branch body on the right.

Another difference from the declarative system is that we do not allow assumptions to be exchanged in the algorithmic system. This is to avoid specifying an algorithm with excessive non-determinism. In fact, it does not matter which order pattern assumptions are treated by the algorithm, however if we were to allow exchange then our completeness theorem would only apply to algorithms which tried every possible order. Fixing a single order forces the completeness theorem to consider possible mismatches between the order used by the algorithm and the order used in a declarative derivation.

The judgment $R \overset{\Rightarrow}{\trianglelefteq} S$ has only three rules: one for when we reach an arrow, and two that non-deterministically decompose an intersection. This judgment is required in order represent the elimination of intersections in a way that accurately reflects what is done in the implementation: in Chapter 3 we instead used elimination rules in the algorithmic judgment for inferable terms, but this results in some unnecessary non-determinism.

The inclusion rules for $\overset{\Rightarrow}{\preceq}$ are based on those for $\preceq$, but with two differences. Firstly, when

the input is a sort R the minimum union is the singleton union R, except when R is empty, in which case the minimum union is the empty union. Secondly, we have a more algorithmic rule for constructor applications.

The judgment $A \overset{\Rightarrow}{\perp} R$ is required when checking products, to allow for uses of subsumption with $R_1 \times R_2 \trianglelefteq S_1 \times S_2$ derived from $R_1 \trianglelefteq \cdot$ or $R_2 \trianglelefteq \cdot$. The judgment only has rules for base types and products: there are no empty refinements for a function types or the unit type $\mathsf{1}$. For a base type a the least refinement is formed by intersecting of all base sorts $r \sqsubset a$, which is then checked for emptiness. For products, we synthesize each empty sort R for the first component, and use these to construct the empty sort $R \times \top$. We then do similarly for the second component.

$$\frac{x{\in}R \text{ in } \Pi}{\Pi \vdash x \overset{\Rightarrow}{\in} R} \qquad \frac{u{\in}R \text{ in } \Pi}{\Pi \vdash u \overset{\Rightarrow}{\in} R} \qquad \frac{c\,S \text{ in } \mathsf{body}(\rho) \qquad \Pi \vdash C \overset{\Leftarrow}{\in} S}{\Pi \vdash c\,C \overset{\Rightarrow}{\in} \rho}$$

$$\frac{}{\Pi \vdash () \overset{\Rightarrow}{\in} 1} \qquad \frac{\Pi \vdash I \overset{\Rightarrow}{\in} S \qquad S \overset{\Rightarrow}{\trianglelefteq} S_1 \to S_2 \qquad \Pi \vdash C \overset{\Leftarrow}{\in} S_1}{\Pi \vdash I\,C \overset{\Rightarrow}{\in} S_2}$$

$$\frac{R \text{ in } L \qquad \Pi, u{\in}R \vdash C \overset{\Leftarrow}{\in} R}{\Pi \vdash \mathbf{fix}\ u.(C{\in}L) \overset{\Rightarrow}{\in} R} \qquad \frac{R \text{ in } L \qquad \Pi \vdash C \overset{\Leftarrow}{\in} R}{\Pi \vdash (C{\in}L) \overset{\Rightarrow}{\in} R}$$

. . . . . .

$$\frac{\Pi, x{\in}R \vdash C \overset{\Leftarrow}{\in} S}{\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R \to S} \qquad \frac{\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R \qquad \Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} S}{\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R \,\&\, S} \qquad \frac{}{\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} \top}$$

$$\frac{R \sqsubset A_1 \times A_2 \qquad A_1 \overset{\Rightarrow}{\perp} S_1 \qquad \Pi \vdash C_1 \overset{\Leftarrow}{\in} S_1 \qquad \Pi \vdash C_2 \overset{\Leftarrow}{\in} \top}{\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R}$$

$$\frac{R \sqsubset A_1 \times A_2 \qquad A_2 \overset{\Rightarrow}{\perp} S_2 \qquad \Pi \vdash C_1 \overset{\Leftarrow}{\in} \top \qquad \Pi \vdash C_2 \overset{\Leftarrow}{\in} S_2}{\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R} \qquad \frac{\Pi \vdash I \overset{\Rightarrow}{\in} R \qquad R \trianglelefteq S}{\Pi \vdash I \overset{\Leftarrow}{\in} S}$$

$$\frac{R \overset{\Rightarrow}{\cong} R_1 \otimes R_2 \qquad \Pi \vdash C_1 \overset{\Leftarrow}{\in} R_1 \qquad \Pi \vdash C_2 \overset{\Leftarrow}{\in} R_2}{\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R} \qquad \frac{\Pi \vdash I \overset{\Rightarrow}{\in} R \qquad \Pi \vdash R \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} S}{\Pi \vdash \mathbf{case}\ I\ \mathbf{of}\ \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} S}$$

. . . . . .

$$\frac{\Pi; P{\in}Z \vdash C \overset{\Leftarrow}{\in} R \qquad \Pi \vdash (Z \backslash P) \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R}{\Pi \vdash Z \bowtie (P \Rightarrow C \mid \Omega^{\mathsf{c}}) \overset{\Leftarrow}{\in} R} \qquad \frac{Z \overset{\Rightarrow}{\trianglelefteq} \bot}{\Pi \vdash Z \bowtie \cdot \overset{\Leftarrow}{\in} R}$$

$$\frac{\Pi;\Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, (P_1,P_2){\in}(Z_1,Z_2) \vdash C \overset{\Leftarrow}{\in} R} \qquad \frac{S \overset{\Rightarrow}{\cong} S_1 \otimes S_2 \quad \Pi;\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, (P_1,P_2){\in}S \vdash C \overset{\Leftarrow}{\in} R}$$

$$\frac{\Pi;\Delta \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, ()\in() \vdash C \overset{\Leftarrow}{\in} R} \qquad \frac{\Pi;\Delta \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, ()\in S \vdash C \overset{\Leftarrow}{\in} R}$$

$$\frac{\Pi;\Delta, P{\in}Z \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, c\,P{\in}c\,Z \vdash C \overset{\Leftarrow}{\in} R} \qquad \frac{c \neq c_2}{\Pi;\Delta, c\,P{\in}c_2\,Z \vdash C \overset{\Leftarrow}{\in} R} \qquad \frac{\Pi;\Delta, c\,P{\in}\mathsf{inv}(\rho) \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, c\,P{\in}\rho \vdash C \overset{\Leftarrow}{\in} R}$$

$$\frac{\Pi;\Delta, P{\in}Z_1 \vdash C \overset{\Leftarrow}{\in} R \quad \Pi;\Delta, P{\in}Z_2 \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, P{\in}Z_1 \sqcup Z_2 \vdash C \overset{\Leftarrow}{\in} R} \qquad \frac{}{\Pi;\Delta, P{\in}\bot \vdash C \overset{\Leftarrow}{\in} R}$$

$$\frac{Z \overset{\Rightarrow}{\preceq} S_1 \sqcup \ldots \sqcup S_n \quad \Pi, x{\in}S_1;\Delta \vdash C \overset{\Leftarrow}{\in} R \;\ldots\; \Pi, x{\in}S_n;\Delta \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\Delta, x{\in}Z \vdash C \overset{\Leftarrow}{\in} R} \qquad \frac{\Pi \vdash C \overset{\Leftarrow}{\in} R}{\Pi;\cdot \vdash C \overset{\Leftarrow}{\in} R}$$

$$\frac{}{R{\to}S \overset{\Rightarrow}{\trianglelefteq} R{\to}S} \qquad \frac{R_1 \overset{\Rightarrow}{\trianglelefteq} S}{R_1 \& R_2 \overset{\Rightarrow}{\trianglelefteq} S} \qquad \frac{R_2 \overset{\Rightarrow}{\trianglelefteq} S}{R_1 \& R_2 \overset{\Rightarrow}{\trianglelefteq} S}$$

$$\frac{\cdot \vdash R \not\trianglelefteq \cdot}{R \overset{\Rightarrow}{\preceq} R} \qquad \frac{\cdot \vdash R \trianglelefteq \cdot}{R \overset{\Rightarrow}{\preceq} \bot} \qquad \frac{Z_1 \overset{\Rightarrow}{\preceq} Y_1 \qquad Z_2 \overset{\Rightarrow}{\preceq} Y_2}{Z_1 \sqcup Z_2 \overset{\Rightarrow}{\preceq} Y_1 \sqcup Y_2} \qquad \frac{}{\bot \overset{\Rightarrow}{\preceq} \bot}$$

$$\frac{}{() \overset{\Rightarrow}{\preceq} 1} \qquad \frac{Z_1 \overset{\Rightarrow}{\preceq} Y_1 \qquad Z_2 \overset{\Rightarrow}{\preceq} Y_2}{(Z_1,Z_2) \overset{\Rightarrow}{\preceq} \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}}$$

$$\frac{Z \overset{\Rightarrow}{\preceq} S_1 \sqcup \ldots \sqcup S_n}{c\,Z \overset{\Rightarrow}{\preceq} \&\{r \mid cS_1 \rightarrowtail r\} \sqcup \ldots \sqcup \&\{r \mid cS_n \rightarrowtail r\}}$$

$$\frac{\&\{r \mid r \sqsubset a\} \trianglelefteq \cdot}{a \overset{\Rightarrow}{\perp} \&\{r \mid r \sqsubset a\}} \qquad \frac{A \overset{\Rightarrow}{\perp} R}{A \times B \overset{\Rightarrow}{\perp} R \times \top^B} \qquad \frac{B \overset{\Rightarrow}{\perp} S}{A \times B \overset{\Rightarrow}{\perp} \top^A \times S}$$

The sort synthesis rule for constructor applications is not quite algorithmic: it does not specify how to determine ρ such that cS in $\mathsf{body}(\rho)$, and in fact syntactically there are an infinite number of possibilities, since ρ may include repetitions of each base sort r. The intention is that only one representative $\rho = r_1 \& \ldots \& r_n$ should be considered for each set $\{r_1, \ldots, r_n\}$ of base sorts refining the type that c belongs to. In the implementation this is further optimized to remove redundant combinations of S and ρ, generally leading to a small number of possibilities.

As in Chapter 2 and Chapter 3 we must extend the typing judgment and declarative sort assignment judgment to terms with annotations. We do this just like before, using the following rules, an erasure function, and appropriate theorems relating annotated terms with their erasures.

$$\frac{R_1 \sqsubset A \quad \ldots \quad R_n \sqsubset A \qquad \Gamma \vdash M : A}{\Gamma \vdash (M \in R_1, \ldots, R_n) : A} \qquad \frac{R \text{ in } L \qquad \Delta \vdash M \in R}{\Delta \vdash (M{\in}L) \in R}$$

$$\begin{array}{rcl}
\|M{\in}L\| & = & \|M\| \\
\|x\| & = & x \\
\|c\,M\| & = & c\,\|M\| \\
\|\lambda x{:}A.M\| & = & \lambda x{:}A.\|M\| \\
\|M\,N\| & = & \|M\|\,\|N\| \\
\|u\| & = & u \\
\|\mathbf{fix}\,u{:}A.M\| & = & \mathbf{fix}\,u{:}A.\|M\| \\
\|(M,N)\| & = & (\|M\|,\|N\|) \\
\|()\| & = & () \\
\|\mathbf{case}\,M\,\mathbf{of}\,\Omega\| & = & \mathbf{case}\,\|M\|\,\mathbf{of}\,\|\Omega\|
\end{array}
\qquad
\begin{array}{rcl}
\|(P{\Rightarrow}M \mid \Omega)\| & = & P{\Rightarrow}\|M\| \mid \|\Omega\| \\
\|\cdot\| & = & \cdot
\end{array}$$

Lemma 6.7.1 (Value Erasure)
If M is a value then $\|M\|$ is a value.

Proof:
By a straightforward induction on the structure of M. □

Lemma 6.7.2 (Typing Erasure)
If $\Gamma \vdash M : A$ then $\Gamma \vdash \|M\| : A$.

Proof:
By a straightforward induction on the structure of the derivation. □

Lemma 6.7.3 (Sorting Erasure)
If $\Pi \vdash M \in R$ then $\Pi \vdash \|M\| \in R$.

Proof:
By a straightforward induction on the structure of the derivation. Each case simply rebuilds the derivation (using the Value Erasure Lemma for those rules which require values), except for the case for the rule for annotations, which follows.

Case: $\dfrac{R \text{ in } L \qquad \Delta \vdash M \in R}{\Delta \vdash (M{\in}L) \in R}$

Then, $\|(M{\in}L)\| = \|M\|$ and $\Delta \vdash \|M\| \in R$ (Ind. Hyp.), as required.

□

6.7.3 Termination of sort checking

Theorem 6.7.4

1. *For any input R the algorithmic judgment*
   $R \overset{\Rightarrow}{\perp} S$ *terminates, producing output S*
   (for possibly many different S).

2. *For any input Z the algorithmic judgment*
   $Z \overset{\Rightarrow}{\preceq} Y$ *terminates, producing a unique output Y.*

3. *For any input A the algorithmic judgment*
   $A \overset{\Rightarrow}{\perp} R$ *terminates, producing output R*
   (for possibly many different R).

Proof: By straightforward inductions on R, Z and A respectively. □

Theorem 6.7.5

1. *For any inputs Π and I the algorithmic judgment*
   $\Pi \vdash I \overset{\Rightarrow}{\in} R$ *terminates, with output R (for possibly many different R).*

2. *For any inputs Π, C and R the algorithmic judgment*
   $\Pi \vdash I \overset{\Leftarrow}{\in} R$ *terminates.*

3. *For any inputs $\Pi, Z, \Omega^{\mathsf{c}}$ and R the algorithmic judgment*
   $\Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$ *terminates.*

4. *For any inputs Π, Δ, C and R the algorithmic judgment*
   $\Pi; \Delta \vdash C \overset{\Leftarrow}{\in} R$ *terminates.*

Proof: By induction on a well order on instances J of the judgments which is defined lexicographically from the following simpler well orders.

1. The syntactic inclusion ordering on the terms I and C and the branches Ω^{c} on the right hand side.

2. An ordering based on which judgment J is an instance of, which is defined by:

   $$\Pi \vdash I \overset{\Rightarrow}{\in} R \quad \text{less than} \quad \Pi \vdash C \overset{\Leftarrow}{\in} R \quad \text{less than} \quad \Pi; \Delta \vdash C \overset{\Leftarrow}{\in} R.$$

   (This part of the whole well-order is never needed for judgments of the form $\Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$.)

3. For instances of the judgment $\Pi \vdash C \overset{\Leftarrow}{\in} R$: the syntactic inclusion ordering on R (to exclude the possibility of an infinite sequence of uses of the &-introduction rule)

4. For instances of the judgment $\Pi; \Delta \vdash C \overset{\Leftarrow}{\in} R$: the lexicographic ordering built from the following well orders.

(a) The ordering according to the sum of the sizes of the patterns P in the context Δ.

(b) The ordering according to the total number of occurrences of base refinements ρ that appear in the pattern sorts in Δ, and which do not appear enclosed by a constructor pattern sort of the form $c\,Z$.

(c) The sum of the sizes of the pattern sorts Z that appear in Δ.

It is straightforward to check that each premise is less that each conclusion according to this well ordering. The following cases are interesting.

Case: $$\frac{\Pi; (\Delta, c\,P{\in}\mathsf{inv}(\rho)) \vdash C \stackrel{\Leftarrow}{\in} R}{\Pi; (\Delta, c\,P{\in}\rho) \vdash C \stackrel{\Leftarrow}{\in} R}$$

Then, part 4b of the ordering is decreased, since ρ appears in the conclusion but not the premise, and $\mathsf{inv}(\rho)$ has the form $c_1\,R_1 \sqcup \ldots \sqcup c_n\,R_n$ and hence has no ρ' appearing except when enclosed by a constructor.

Case: $$\frac{\Pi; (\Delta, P{\in}Z) \vdash C \stackrel{\Leftarrow}{\in} R}{\Pi; (\Delta, c\,P{\in}c\,Z) \vdash C \stackrel{\Leftarrow}{\in} R}$$

Then, part 4a of the ordering is decreased. Hence, the whole ordering is decreased (even though part 4b is increased). □

6.7.4 Soundness of sort checking

As in Chapter 2 and Chapter 3, the proof of soundness of sort checking is relatively simple: the derivations of the algorithmic system correspond to a fragment of the declarative system. We start with the following lemma, which has similarities to the distributivity property required of signatures in Section 3.2.

Lemma 6.7.6 (Distributivity of $\rightarrowtail$)

1. *If $c\,R_1 \rightarrowtail s_1$ and $c\,R_2 \rightarrowtail s_2$ then $c(R_1 \& R_2) \rightarrowtail (s_1 \& s_2)$.*
2. *If $a = D$ and $c\,B$ in D then $c\top^B \rightarrowtail \top^a$.*

Proof:

For part 1:

$\mathsf{body}(s_1 \& s_2) = \sqcup\{c(S_1 \& S_2) \mid c\,S_1 \text{ in } \mathsf{body}(s_1), c\,S_2 \text{ in } \mathsf{body}(s_2)\}$	Def. body
$c\,S_1'$ in $\mathsf{body}(s_1)$ with $R_1 \trianglelefteq S_1'$	Def. $\rightarrowtail$
$c\,S_2'$ in $\mathsf{body}(s_2)$ with $R_2 \trianglelefteq S_2'$	Def. $\rightarrowtail$
$c(S_1' \& S_2')$ in $\mathsf{body}(s_1 \& s_2)$	
$R_1 \& R_2 \trianglelefteq S_1' \& S_2'$	& is g.l.b. (5.8.11)
$c(R_1 \& R_2) \rightarrowtail (s_1 \& s_2)$	Def. $\rightarrowtail$

For part 2:

$\mathsf{body}(\top^a) = \sqcup\{c\top^A \mid cA \text{ in } D\}$ with $a = D$ in Σ	Def. body
cB in D	Assumed
$c\top^B$ in $\mathsf{body}(\top^a)$	
$\top^B \trianglelefteq \top^B$	Reflexivity (5.8.11)
$c(\top^B) \rightarrowtail (\top^a)$	Def. $\rightarrowtail$

□

The statement of the soundness theorem has one part for each judgment in the algorithm. Each part is designed to be general enough to support the inductive proof which follows.

Theorem 6.7.7 (Soundness of Sort Checking)

1. *If* $\Pi \vdash I \overset{\Rightarrow}{\in} R$ *then* $\Pi \vdash I \in R$.
2. *If* $\Pi \vdash C \overset{\Leftarrow}{\in} S$ *then* $\Pi \vdash C \in S$.
3. *If* $\Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$ *then* $\Pi \vdash Z \bowtie \Omega \in R$.
4. *If* $\Pi; \Delta \vdash C \overset{\Leftarrow}{\in} R$ *then* $(\Pi, \Delta) \vdash C \in R$.
5. *If* $R \overset{\Rightarrow}{\trianglelefteq} S$ *then* $R \trianglelefteq S$.
6. *If* $Z \overset{\Rightarrow}{\preceq} Y$ *then* $Z \preceq Y$.
7. *If* $A \overset{\Rightarrow}{\perp} R$ *then* $R \trianglelefteq \bullet$.

Proof:
By induction on the structure of the derivations. Most cases simply rebuild the derivation, using the corresponding rule in the declarative system. We show only a few such cases; the remaining ones are similar. We also show the cases that require more than just rebuilding the derivation.

We have the following cases for part 1 which require more than simply rebuilding the derivation.

Case: $\dfrac{(c\,S) \text{ in } \mathsf{body}(\rho) \qquad \Pi \vdash C \overset{\Leftarrow}{\in} S}{\Pi \vdash c\,C \overset{\Rightarrow}{\in} \rho}$

$\Pi \vdash C \in S$	Ind. Hyp.
$S \trianglelefteq S$	Reflexivity (5.8.11)
$cS \rightarrowtail \rho$	Rule for $\rightarrowtail$
$\Pi \vdash cC \in \rho$	

Case: $\dfrac{\Pi \vdash I \overset{\Rightarrow}{\in} S \qquad S \overset{\Rightarrow}{\trianglelefteq} S_1 \to S_2 \qquad \Pi \vdash C \overset{\Leftarrow}{\in} S_1}{\Pi \vdash I\, C \overset{\Rightarrow}{\in} S_2}$

$\Pi \vdash I \in S$	Ind. Hyp.
$S \trianglelefteq S_1 \to S_2$	Ind. Hyp.
$\Pi \vdash I \in S_1 \to S_2$	Subsumption Rule
$\Pi \vdash C \in S_1$	Ind. Hyp.
$\Pi \vdash I\, C \in S_2$	Rule for appl.

The following are some of the cases for part 1 which simply rebuild the derivation (the others are similar).

Case: $\dfrac{R \text{ in } L \qquad \Pi, u{\in}R \vdash C \overset{\Leftarrow}{\in} R}{\Pi \vdash \mathbf{fix}\, u.(C{\in}L) \overset{\Rightarrow}{\in} R}$

$\Pi, u{\in}R \vdash C \in R$	Ind. Hyp.
$\Pi, u{\in}R \vdash (C{\in}L) \in R$	Rule for Annot.
$\Pi \vdash \mathbf{fix}\, u.(C{\in}L) \in R$	Rule for **fix**

Case: $\dfrac{R \text{ in } L \qquad \Pi \vdash C \overset{\Leftarrow}{\in} R}{\Pi \vdash (C{\in}L) \overset{\Rightarrow}{\in} R}$

$\Pi \vdash C \in R$	Ind. Hyp.
$\Pi \vdash (C{\in}L) \in R$	Rule for Annot.

We have the following cases for part 2 (and some additional cases which simply rebuild the derivation).

Case: $\dfrac{R \overset{\Rightarrow}{\cong} R_1 \otimes R_2 \qquad \Pi \vdash C_1 \overset{\Leftarrow}{\in} R_1 \qquad \Pi \vdash C_2 \overset{\Leftarrow}{\in} R_2}{\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R}$

$\Pi \vdash C_1 \in R_1$	Ind. Hyp.
$\Pi \vdash C_2 \in R_2$	Ind. Hyp.
$\Pi \vdash (C_1, C_2) \in R_1 \times R_2$	Rule for Pairs
$R_1 \times R_2 \trianglelefteq R$	Lemma 5.6.8

Case: $\dfrac{R \sqsubset A_1 \times A_2 \quad A_1 \overset{\Rightarrow}{\perp} S_1 \quad \Pi \vdash C_1 \overset{\Leftarrow}{\in} S_1 \quad \Pi \vdash C_2 \overset{\Leftarrow}{\in} \top}{\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R}$

$\Pi \vdash C_1 \in S_1$	Ind. Hyp.
$\Pi \vdash C_2 \in \top$	Ind. Hyp.
$\Pi \vdash (C_1, C_2) \in S_1 \times \top$	Rule for Pairs
$S_1 \trianglelefteq \cdot$	Ind. Hyp.
$[\![S_1]\!] \subseteq \{\}$	Lemma 5.8.10
$[\![S_1 \times \top]\!] \subseteq \{\}$	Def. $[\![\cdot]\!]$
$S_1 \times \top \trianglelefteq R$	Lemma 5.8.10
$\Pi \vdash (C_1, C_2) \in R$	Subsumption Rule

Case: $\dfrac{R \sqsubset A_1 \times A_2 \quad A_2 \overset{\Rightarrow}{\perp} S_2 \quad \Pi \vdash C_1 \overset{\Leftarrow}{\in} \top \quad \Pi \vdash C_2 \overset{\Leftarrow}{\in} S_2}{\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R}$

Dual to the previous case.

Case: $\dfrac{\Pi \vdash I \overset{\Rightarrow}{\in} R \quad R \trianglelefteq S}{\Pi \vdash I \overset{\Leftarrow}{\in} S}$

$\Pi \vdash I \in R$	Ind. Hyp.
$\Pi \vdash I \in S$	Subsumption Rule

Both cases for part 3 simply rebuild the derivation.

Case: $\dfrac{\Pi; P \in Z \vdash C \overset{\Leftarrow}{\in} R \qquad \Pi \vdash (Z \backslash P) \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R}{\Pi \vdash Z \bowtie (P \Rightarrow C \mid \Omega^{\mathsf{c}}) \overset{\Leftarrow}{\in} R}$

$\Pi, P \in Z \vdash C \in R$	Ind. Hyp.
$\Pi \vdash (Z \backslash P) \bowtie \Omega^{\mathsf{c}} \in R$	Ind. Hyp.
$\Pi \vdash Z \bowtie (P \Rightarrow C \mid \Omega^{\mathsf{c}}) \in R$	Rule

Case: $\dfrac{Z \overset{\Rightarrow}{\preceq} \bot}{\Pi \vdash_\Sigma Z \bowtie \cdot \overset{\Leftarrow}{\in} R}$

$Z \preceq \bot$	Ind. Hyp.
$\Pi \vdash_\Sigma Z \bowtie \cdot \in R$	Rule

We have the following cases for part 4 (and some additional cases that simply rebuild the derivation).

Case: $\dfrac{Z \overset{\Rightarrow}{\preceq} S_1 \sqcup \ldots \sqcup S_n \quad (\Pi, x{\in}S_1); \Delta \vdash C \overset{\Leftarrow}{\in} R \ \ldots \ (\Pi, x{\in}S_n); \Delta \vdash C \overset{\Leftarrow}{\in} R}{\Pi; (\Delta, x{\in}Z) \vdash C \overset{\Leftarrow}{\in} R}$

$Z \preceq S_1 \sqcup \ldots S_n$	Ind. Hyp.
$\Pi, x{\in}S_1, \Delta \vdash C \in R$ and $\ldots$ and	
$\Pi, x{\in}S_n, \Delta \vdash C \in R$	Ind. Hyp. (for each)
$\Pi, x{\in}(S_1 \sqcup \ldots \sqcup S_n), \Delta \vdash C \in R$	Rule for $\sqcup$, repeatedly
$\Pi, (\Delta, x{\in}Z) \vdash C \in R$	Left Subsumption Rule

Case: $\dfrac{\Pi \vdash C \overset{\Leftarrow}{\in} R}{\Pi; \cdot \vdash C \overset{\Leftarrow}{\in} R}$

$\Pi \vdash C \in R$	Ind. Hyp.
$(\Pi, \cdot) \vdash C \in R$	Since $(\Pi, \cdot) = \Pi$

We have the following cases for Part 5.

Case: $\dfrac{}{R{\to}S \overset{\Rightarrow}{\trianglelefteq} R{\to}S}$: By reflexivity (Lemma 5.8.11).

Case: $\dfrac{R_1 \overset{\Rightarrow}{\trianglelefteq} S}{R_1 \& R_2 \overset{\Rightarrow}{\trianglelefteq} S}$

$R_1 \trianglelefteq S$	Ind. hyp.
$R_1 \& R_2 \trianglelefteq R_1$	& is g.l.b. (5.8.11)
$R_1 \& R_2 \trianglelefteq S$	Transitivity (5.8.11)

Case: $\dfrac{R_2 \overset{\Rightarrow}{\trianglelefteq} S}{R_1 \& R_2 \overset{\Rightarrow}{\trianglelefteq} S}$

Dual to the previous case.

We have the following cases for Part 6.

Case: $\dfrac{\cdot \vdash R \trianglelefteq \cdot}{R \overset{\Rightarrow}{\preceq} \bot}$

$R \preceq \bot$	Rule for Empty Sort

Case: $\dfrac{\cdot \vdash R \not\trianglelefteq \cdot}{R \overset{\Rightarrow}{\preceq} R}$

$R \trianglelefteq R$	Reflexivity (5.8.11)
$R \preceq R$	Rule for Subsort

Case: $\dfrac{Z_1 \stackrel{\Rightarrow}{\preceq} Y_1 \qquad Z_2 \stackrel{\Rightarrow}{\preceq} Y_2}{Z_1 \sqcup Z_2 \stackrel{\Rightarrow}{\preceq} Y_1 \sqcup Y_2}$

$Z_1 \preceq Y_1$	Ind. Hyp
$Z_2 \preceq Y_2$	Ind. Hyp
$Z_1 \sqcup Z_2 \preceq Y_1 \sqcup Y_2$	Rule for $\sqcup$

Case: $\dfrac{}{() \stackrel{\Rightarrow}{\preceq} 1}$

$() \preceq 1$	Rule for $()$

Case: $\dfrac{Z_1 \stackrel{\Rightarrow}{\preceq} Y_1 \qquad Z_2 \stackrel{\Rightarrow}{\preceq} Y_2}{(Z_1, Z_2) \stackrel{\Rightarrow}{\preceq} \sqcup\{R_1 \times R_2 \,|\, R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}}$

$Z_1 \preceq Y_1$	Ind. Hyp
$Z_2 \preceq Y_2$	Ind. Hyp
$(Z_1, Z_2) \preceq \sqcup\{R_1 \times R_2 \,|\, R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}$	Rule for (Z_1,Z_2)

Case: $\dfrac{Z \stackrel{\Rightarrow}{\preceq} S_1 \sqcup \ldots \sqcup S_n}{c\,Z \stackrel{\Rightarrow}{\preceq} \&\{r \,|\, cS_1 \rightarrowtail r\} \sqcup \ldots \sqcup \&\{r \,|\, cS_n \rightarrowtail r\}}$

$Z \preceq S_1 \sqcup \ldots \sqcup S_n$	Ind. Hyp
For each S_i:	
$cS_i \rightarrowtail r$ for each r in $\&\{r \,|\, cS_i \rightarrowtail r\}$	
$c(S_i \& \ldots \& S_i) \rightarrowtail \&\{r \,|\, cS_i \rightarrowtail r\}$	Dist. of $\rightarrowtail$ (6.7.6)
$cS_i \rightarrowtail \&\{r \,|\, cS_i \rightarrowtail r\}$	$\&$ is g.l.b (5.8.11)
$c\,Z \preceq \&\{r \,|\, cS_1 \rightarrowtail r\} \sqcup \ldots \sqcup \&\{r \,|\, cS_n \rightarrowtail r\}$	Rule for $c\,Z$

We have the following cases for Part 7.

Case: $\dfrac{\&\{r \,|\, r \sqsubset a\} \trianglelefteq \cdot}{a \stackrel{\Rightarrow}{\perp} \&\{r \,|\, r \sqsubset a\}}$ Immediate.

Case: $\dfrac{A \stackrel{\Rightarrow}{\perp} R}{A \times B \stackrel{\Rightarrow}{\perp} R \times \top^B}$ Then $R \trianglelefteq \cdot$ (ind. hyp.), so $R \times \top^B \trianglelefteq \cdot$ (rules for $\times$).

Case: $\dfrac{B \stackrel{\Rightarrow}{\perp} S}{A \times B \stackrel{\Rightarrow}{\perp} \top^A \times S}$ Dual to the previous case.

□

6.7.5 Completeness of sort checking

As in Chapter 2 and Chapter 3, demonstrating completeness of the sort checking algorithm is more difficult than soundness. Further, some new complications result from the flexibility

allowed in the declarative left rules for pattern sort assumptions.

We handle these complications by extending the technique used in the previous completeness proofs: proving an inversion lemma corresponding to each rule in the algorithmic system. These lemmas roughly state that if there is a declarative derivation with a conclusion matching the form of the algorithmic rule, then there are derivations for the premises of the rule. Since this implies that there is a derivation ending with an instance of the rule, the inversion lemmas for left rules are related to the permutability of left rules in sequent calculi.

We will require the following lemma in the proof of the inversion lemma for the left subsumption rule.

Lemma 6.7.8 (Completeness of $\overset{\Rightarrow}{\preceq}$)
If $Z \preceq Y'$ then $Z \overset{\Rightarrow}{\preceq} Y$
with each S in Y satisfying $S \trianglelefteq S'$ for some S' in Y'.

Proof: By induction on structure of the derivation of $Z \preceq Y'$.

Case: $\dfrac{\cdot \vdash_\Sigma R \trianglelefteq S}{R \preceq S}$

If $R \trianglelefteq \cdot$ then:

$R \overset{\Rightarrow}{\preceq} \bot$ — Rule for empty sort

Second part holds vacuously — No S in $\bot$

Otherwise $R \not\trianglelefteq \cdot$, and then:

$R \overset{\Rightarrow}{\preceq} R$ — Rule for non-empty sort

$R \trianglelefteq S$ — Assumed

Case: $\dfrac{\cdot \vdash_\Sigma R \trianglelefteq \cdot}{R \preceq \bot}$

$R \overset{\Rightarrow}{\preceq} \bot$ — Rule for empty sort

Second part holds vacuously — No S in $\bot$

Case: $\dfrac{Z_1 \preceq Y_1' \qquad Z_2 \preceq Y_2'}{Z_1 \sqcup Z_2 \preceq \sqcup\{R \,|\, R \text{ in } Y_1' \text{ or } Y_2'\}}$

$Z_1 \overset{\Rightarrow}{\preceq} Y_1$ with S_1 in Y_1 implies
$S_1 \trianglelefteq S_1'$ for some S_1' in Y_1'. — Ind. Hyp.

$Z_2 \overset{\Rightarrow}{\preceq} Y_2$ with S_2 in Y_2 implies
$S_2 \trianglelefteq S_2'$ for some S_2' in Y_2'. — Ind. Hyp.

$Z_1 \sqcup Z_2 \overset{\Rightarrow}{\preceq} Y_1 \sqcup Y_2$ — Rule for $\sqcup$

S in $Y_1 \sqcup Y_2$ implies:

S in Y_1 or S in Y_2 — Def. "in"

$S \trianglelefteq S'$ for some S' in Y_1'

$\quad$ or $\;S \trianglelefteq S'$ for some S' in Y_2' — Instantiating above

$S \trianglelefteq S'$ for some S' in $\sqcup\{R \mid R \text{ in } Y_1' \text{ or } Y_2'\}$ — Def. "in"

Case: $\dfrac{}{\bot \preceq \bot}$

$\bot \overset{\Rightarrow}{\preceq} \bot$ — Rule for $\bot$

Second part holds vacuously — No S in $\bot$

Case: $\dfrac{}{() \preceq 1}$

$() \overset{\Rightarrow}{\preceq} 1$ — Rule for $()$

$() \trianglelefteq ()$ — Reflexivity (5.8.11)

Case: $\dfrac{Z_1 \preceq Y_1' \qquad Z_2 \preceq Y_2'}{(Z_1, Z_2) \preceq \sqcup\{R_1' \times R_2' \mid R_1' \text{ in } Y_1', R_2' \text{ in } Y_2'\}}$

$Z_1 \overset{\Rightarrow}{\preceq} Y_1$ with S_1 in Y_1 implies

$\quad S_1 \trianglelefteq S_1'$ for some S_1' in Y_1' — Ind. Hyp.

$Z_2 \overset{\Rightarrow}{\preceq} Y_2$ with S_2 in Y_2 implies

$\quad S_2 \trianglelefteq S_2'$ for some S_2' in Y_2' — Ind. Hyp.

$(Z_1,Z_2) \overset{\Rightarrow}{\preceq} \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}$ — Rule for $\sqcup$

$R_1 \times R_2$ in $\sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1, R_2 \text{ in } Y_2\}$ implies:

$\quad R_1$ in Y_1 and R_2 in Y_2 — Def. "in"

$\quad R_1 \trianglelefteq R_1'$ for some R_1' in Y_1' — Instantiating above

$\quad R_2 \trianglelefteq R_2'$ for some R_2' in Y_2' — Instantiating above

$\quad R_1 \times R_2 \trianglelefteq R_1' \times R_2'$ — Rules for $\times$

$\quad R_1' \times R_2'$ in $\sqcup\{R_1' \times R_2' \mid R_1' \text{ in } Y_1', R_2' \text{ in } Y_2'\}$ — Def. "in"

Case: $\dfrac{Z \preceq R_1 \sqcup \ldots \sqcup R_n \qquad c\,R_1 \rightarrowtail \rho_1 \;\; \ldots \;\; c\,R_n \rightarrowtail \rho_n}{c\,Z \preceq \rho_1 \sqcup \ldots \sqcup \rho_n}$

$Z \overset{\Rightarrow}{\preceq} S_1 \sqcup \ldots \sqcup S_m$ with $S_i \trianglelefteq R_j$ for some R_j for each S_i — Ind. Hyp.

$c\,Z \overset{\Rightarrow}{\preceq} \&\{r \mid c\,S_1 \rightarrowtail r\} \sqcup \ldots \sqcup \&\{r \mid c\,S_m \rightarrowtail r\}$ — Rule for $c\,Z$

For each $\&\{r \mid c\,S_i \rightarrowtail r\}$:

$\quad S_i \trianglelefteq R_j$ — Instantiating above

$\quad c\,R_j \rightarrowtail \rho_j$ — Assumed

$\quad c\,R_j'$ in $\mathsf{body}(\rho_j)$ with $R_j \trianglelefteq R_j'$ — Inv. on $\rightarrowtail$

$\quad R_j' = S_1' \& \ldots \& S_h'$

$\qquad$ where $\rho_j = r_1 \& \ldots \& r_h$

and $c\,S'_1$ in $\mathsf{body}(r_1), \ldots, c\,S'_h$ in $\mathsf{body}(r_h)$	Def. body, repeatedly
$R'_j \trianglelefteq S'_k$ for each S'_k	& is g.l.b (5.8.11)
$R_j \trianglelefteq S'_k$ for each S'_k	Transitivity (5.8.11)
$S_i \trianglelefteq S'_k$ for each S'_k	Transitivity (5.8.11)
$c\,S_i \rightarrowtail r_k$ for each r_k	Rule for $\rightarrowtail$
$\&\{r \mid c\,S_i \rightarrowtail r\} \trianglelefteq \rho_j$	& is g.l.b (5.8.11)
	$\square$

We have three kinds of inversion lemmas: those corresponding to rules for synthesizing sorts of terms, those corresponding to rules for checking terms against sorts, and those corresponding left rules. For the first two, we restrict the context to be of the form Π since this is the form in the corresponding algorithmic rules: our completeness proof will follow the steps of the algorithm, hence will only require these lemmas for this form of context. We have no inversion lemmas corresponding to the rules for the judgment $\Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$: standard inversion suffices for this judgment.

Lemma 6.7.9 (Inversion Lemmas)

1. (a) *If* $\Pi \vdash x \in R$ *then* $x \in S$ *in* Π *for some* $S \trianglelefteq R$.
   (b) *If* $\Pi \vdash u \in R$ *then* $u \in S$ *in* Π *for some* $S \trianglelefteq R$.
   (c) *If* $\Pi \vdash c\,M \in \rho$ *then* $c\,S$ *in* $\mathsf{body}(\rho')$ *and* $\Pi \vdash M \in S$ *and* $\rho' \trianglelefteq \rho$ *for some* S *and* ρ'.
   (d) *If* $\Pi \vdash M\,N \in R$ *then* $\Pi \vdash M \in R_2 \to R$ *and* $\Pi \vdash N \in R_2$ *for some* R_2.
   (e) *If* $\Pi \vdash (M \in L) \in R$ *then* $\Pi \vdash M \in S$ *for some* S *in* L *with* $S \trianglelefteq R$.
   (f) *If* $\Pi \vdash () \in R$ *then* $\mathsf{1} \trianglelefteq R$.
   (g) *If* $\Pi \vdash \mathbf{fix}\,u.(M \in L) \in R$ *then* $\Pi, u \in S \vdash M \in S$ *for some* S *in* L *with* $S \trianglelefteq R$.

2. (a) *If* $\Pi \vdash \lambda x.M \in R$ *and* $R \trianglelefteq S_1 \to S_2$ *then* $\Pi, x \in S_1 \vdash M \in S_2$.
   (b) *If* $\Pi \vdash (M_1, M_2) \in R$ *and* $R \overset{\Rightarrow}{\cong} R_1 \otimes R_2$ *then* $\Pi \vdash M_1 \in S_1$ *and* $\Pi \vdash M_2 \in S_2$ *for some* S_1 *and* S_2 *with* $S_1 \times S_2 \trianglelefteq R_1 \times R_2$.
   (c) *If* $\Pi \vdash \mathbf{case}\,M\,\mathbf{of}\,\Omega \in R$ *and* $R \trianglelefteq R'$ *then* $\Pi \vdash M \in S$ *and* $\Pi \vdash S \bowtie \Omega \in R'$ *for some* S.

3. *Each of the following hold when* J *has one of the two forms* $M \in R$ *and* $Z \bowtie \Omega \in R$, *i.e. they hold for the two forms of judgment* $\Delta \vdash M \in R$ *and* $\Delta \vdash Z \bowtie \Omega \in R$.
   (a) *If* $\Delta, P \in Z_1 \sqcup Z_2 \vdash J$ *then* $\Delta, P \in Z_1 \vdash J$ *and* $\Delta, P \in Z_2 \vdash J$.
   (b) *If* $\Delta, (P_1, P_2) \in S \vdash J$ *then* $S \overset{\Rightarrow}{\cong} S_1 \otimes S_2$ *and* $\Delta, P_1 \in S_1, P_2 \in S_2 \vdash J$.
   (c) *If* $\Delta, (P_1, P_2) \in (Z_1, Z_2) \vdash J$ *then* $\Delta, P_1 \in Z_1, P_2 \in Z_2 \vdash J$.
   (d) *If* $\Delta, () \in () \vdash J$ *then* $\Delta \vdash J$.

(e) If $\Delta, ()\in S \vdash J$ *then* $\Delta \vdash J$.

(f) If $\Delta, cP\in cZ \vdash J$ *then* $\Delta, P\in Z \vdash J$.

(g) If $\Delta, cP\in\rho \vdash J$ *then* $\Delta, cP\in \mathsf{inv}(\rho) \vdash J$.

(h) If $\Delta, x\in Z \vdash J$
then $Z \overset{\Rightarrow}{\preceq} Y$ *and for each* S *in* Y *we have* $\Delta, x\in S \vdash J$.

Proof: Each part is proved by a separate induction on the structure of the given sort assignment derivation. (Although some parts make use of earlier parts.)

Parts *1a-1g* and *2a-2c* each have one case for the rule involving the corresponding term construct, and one case for the subsumption rule. There are additional cases for the & and $\top^A$ introduction rules for those parts where the term may be a value. The remaining cases cannot occur, since they involve rules which have a different term construct in the conclusion.

Parts *3c-3h* each have only one truly interesting case: that for left subsumption rule applied to the last assumption. They also have one case for the corresponding declarative left rule applied to the last assumption, for which the required result is immediate. Additionally they have one case for each right rule, one for each of the rules for $\Delta \vdash Z \bowtie \Omega \overset{\Leftarrow}{\in} R$, one case for each left rule applied to a different assumption. These cases simply apply the induction hypothesis as appropriate and then rebuild the derivation.

1. (a) If $\Pi \vdash x \in R$ then $x\in S$ in Π for some $S \trianglelefteq R$.

Case: $\dfrac{x\in R \text{ in } \Pi}{\Pi \vdash x \in R}$

$R \trianglelefteq R$	Reflexivity (5.8.11)

Case: $\dfrac{\Pi \vdash x \in S \qquad S \trianglelefteq R}{\Pi \vdash x \in R}$

$x\in S'$ in Π with $S' \trianglelefteq S$	Ind. Hyp.
$S' \trianglelefteq R$	Transitivity (5.8.11)

Case: $\dfrac{\Pi \vdash x \in R_1 \qquad \Pi \vdash x \in R_2}{\Pi \vdash x \in R_1 \& R_2}$

$x\in S_1$ in Π with $S_1 \trianglelefteq R_1$	Ind. Hyp.
$x\in S_2$ in Π with $S_2 \trianglelefteq R_2$	Ind. Hyp.
$S_1 = S_2$	x occurs only once in Π
$S_1 \trianglelefteq R_1 \,\&\, R_2$	& is g.l.b (5.8.11)

Case: $\dfrac{}{\Pi \vdash x \in \top^A}$

$\Pi \sqsubset \Gamma$ with $\Gamma \vdash x \in A$	Well-formedness of $\Pi \vdash x \in \top^A$
$x{:}A$ in Γ	Inversion
$x{\in}S$ in Π with $S \sqsubset A$	Def. $\Pi \vdash\sqsubset \Gamma$
$\Pi \vdash x \in S$	Rule for x
$S \trianglelefteq \top^A$	$\top$ is maximum (5.8.11)

(b) If $\Pi \vdash u \in R$ then $u{\in}S$ in Π for some $S \trianglelefteq R$.

As for the first two cases of the previous part, replacing x by u. The latter two cases are not needed for u, since it is not a value.

(c) If $\Pi \vdash c\,M \in \rho$ then $c\,S$ in $\mathsf{body}(\rho')$ and $\Pi \vdash M \in S$ and $\rho' \trianglelefteq \rho$ for some S and ρ'.

Case: $\dfrac{c\,S \rightarrowtail \rho \qquad \Pi \vdash M \in S}{\Pi \vdash c\,M \in \rho}$

$c\,S'$ in $\mathsf{body}(\rho)$ with $S \trianglelefteq S'$	Def. $\rightarrowtail$
$\Pi \vdash M \in S'$	Subsumption Rule

Case: $\dfrac{\Pi \vdash c\,M \in \rho_2 \qquad \rho_2 \trianglelefteq \rho}{\Pi \vdash c\,M \in \rho}$

$c\,S$ in $\mathsf{body}(\rho')$	
with $\Pi \vdash M \in S$ and $\rho' \trianglelefteq \rho_2$	Ind. Hyp.
$\rho' \trianglelefteq \rho$	Transitivity (5.8.11)

Case: $\dfrac{\Pi \vdash c\,V \in \rho_1 \qquad \Pi \vdash c\,V \in \rho_2}{\Pi \vdash c\,V \in \rho_1 \& \rho_2}$

$c\,S_1$ in $\mathsf{body}(\rho_1')$	
with $\Pi \vdash V \in S_1$ and $\rho_1' \trianglelefteq \rho_1$	Ind. Hyp.
$c\,S_2$ in $\mathsf{body}(\rho_2')$	
with $\Pi \vdash V \in S_2$ and $\rho_2' \trianglelefteq \rho_2$	Ind. Hyp.
$c(S_1 \& S_2)$ in $\mathsf{body}(\rho_1' \& \rho_2')$	Def. $\mathsf{body}(\cdot)$
$\Pi \vdash V \in S_1 \& S_2$	&-intro.
$\rho_1' \& \rho_2' \trianglelefteq \rho_1 \& \rho_2$	& is g.l.b (5.8.11)

Case: $\dfrac{}{\Pi \vdash cV \in \top^a}$

$\Pi \sqsubset \Gamma$ with $\Gamma \vdash cV : a$	Well-form. of $\Pi \vdash cV \in \top^a$
$\Gamma \vdash V : B$ with cB in D and $a = D$ in Σ	Inversion
$c\top^B$ in $\mathsf{body}(\top^a)$	Def. $\mathsf{body}(\cdot)$
$\Pi \vdash V \in \top^B$	Rule for Constr. App.
$\top^a \trianglelefteq \top^a$	Reflexivity (5.8.11)

(d) If $\Pi \vdash M\,N \in R$ then $\Pi \vdash M \in R_2 \to R$ and $\Pi \vdash N \in R_2$ for some R_2.

Case: $\dfrac{\Pi \vdash M \in S \to R \qquad \Pi \vdash N \in S}{\Pi \vdash M\,N \in R}$

Immediate, choosing $R_2 = S$.

Case: $\dfrac{\Pi \vdash M\,N \in S \qquad S \trianglelefteq R}{\Pi \vdash M\,N \in R}$

$\Pi \vdash M \in S_2 \to S$ with $\Pi \vdash N \in S_2$	Ind. Hyp.
$S_2 \trianglelefteq S_2$	Reflexivity (5.8.11)
$S_2 \to S \trianglelefteq S_2 \to R$	Rule for $\to$
$\Pi \vdash M \in S_2 \to R$	Subsumption Rule

(e) If $\Pi \vdash (M{\in}L) \in R$ then $\Pi \vdash M \in S$ for some S in L with $S \trianglelefteq R$.

Case: $\dfrac{R \text{ in } L \qquad \Pi \vdash M \in R}{\Pi \vdash (M{\in}L) \in R}$

$R \trianglelefteq R$	Reflexivity (5.8.11)

Case: $\dfrac{\Pi \vdash (M{\in}L) \in R' \qquad R' \trianglelefteq R}{\Pi \vdash (M{\in}L) \in R}$

$\Pi \vdash M \in S$ with S in L and $S \trianglelefteq R'$	Ind. Hyp.
$S \trianglelefteq R$	Transitivity (5.8.11)

(f) If $\Pi \vdash () \in R$ then $\mathsf{1} \trianglelefteq R$.

This can be proved inductively, similar to the other parts, but there is also a simpler non-inductive proof, which follows.

$R \sqsubset A$ and $\Pi \sqsubset \Gamma$ with $\Gamma \vdash () : A$	Well-form. of $\Pi \vdash () \in R$
$A = \mathsf{1}$	Inv. on $\Gamma \vdash () : A$
$R \sqsubset \mathsf{1}$	Substituting into $R \sqsubset A$
$\mathsf{1} \sqsubset \mathsf{1}$	Validity Rule for $\mathsf{1}$
$\mathsf{1} \trianglelefteq R$	Inclusion Rule for $\mathsf{1}$

(g) If $\Pi \vdash \mathbf{fix}\, u.(M \in L) \in R$
then $\Pi, u \in S \vdash M \in S$ for some S in L with $S \trianglelefteq R$.

Case: $\dfrac{\Pi, u \in R \vdash (M \in L) \in R}{\Pi \vdash \mathbf{fix}\, u.(M \in L) \in R}$

$\Pi, u \in R \vdash M \in S$ with S in L and $S \trianglelefteq R$	By Part 1e
$\Pi, u \in S \vdash u \in R$	Rule for u, Subsumption
$\Pi, u \in S \vdash M \in S$	Exp. Subst. Lem. (6.6.2)

Case: $\dfrac{\Pi \vdash \mathbf{fix}\, u.(M \in L) \in R' \qquad R' \trianglelefteq R}{\Pi \vdash \mathbf{fix}\, u.(M \in L) \in R}$

$\Pi, u \in S \vdash M \in S$ with S in L and $S \trianglelefteq R'$	Ind. Hyp.
$S \trianglelefteq R$	Transitivity (5.8.11)

2. (a) If $\Pi \vdash \lambda x.M \in R$ and $R \trianglelefteq S_1 \to S_2$ then $\Pi, x \in S_1 \vdash M \in S_2$.

Case: $\dfrac{\Pi, x \in R_1 \vdash M \in R_2}{\Pi \vdash \lambda x.M \in R \in R_1 \to R_2}$

$R_1 \to R_2 \trianglelefteq S_1 \to S_2$	Assumed
$S_1 \trianglelefteq R_1$ and $R_2 \trianglelefteq S_2$	Inversion
$\Pi, x \in S_1 \vdash x \in R_1$	Rule for x, Subsumption
$\Pi, x \in S_1 \vdash M \in R_2$	Pat. Subst. Lem. (6.6.13)
$\Pi, x \in S_1 \vdash M \in S_2$	Subsumption Rule

Case: $\dfrac{\Pi \vdash \lambda x.M \in R' \qquad R' \trianglelefteq R}{\Pi \vdash \lambda x.M \in R}$

$R' \trianglelefteq S_1 \to S_2$	Transitivity (5.8.11)
$\Pi, x \in S_1 \vdash M \in S_2$	Ind. Hyp.

Case: $\dfrac{\Pi \vdash \lambda x.M \in R_1 \qquad \Pi \vdash \lambda x.M \in R_2}{\Pi \vdash \lambda x.M \in R_1 \& R_2}$

$R_1 \& R_2 \trianglelefteq S_1 \to S_2$ Assumed

$R_1 \trianglelefteq S_1 \to S_2$ or $R_1 \trianglelefteq S_1 \to S_2$ Inversion

If $R_1 \trianglelefteq S_1 \to S_2$:

$\Pi, x{\in}S_1 \vdash M \in S_2$ Ind. Hyp.

Otherwise $R_2 \trianglelefteq S_1 \to S_2$:

$\Pi, x{\in}S_1 \vdash M \in S_2$ Ind. Hyp.

Case: $\dfrac{}{\Pi \vdash \lambda x.M \in \top^{A_1 \to A_2}}$

$\top^{A_1 \to A_2} \trianglelefteq S_1 \to S_2$ Assumed

Cannot occur, no rule matches this conclusion.

(b) If $\Pi \vdash (M_1,M_2) \in R$ then $\Pi \vdash M_1 \in S_1$ and $\Pi \vdash M_2 \in S_2$ for some S_1 and S_2 with $S_1 \times S_2 \trianglelefteq R$.

Case: $\dfrac{\Pi \vdash M_1 \in R_1 \qquad \Pi \vdash M_2 \in R_2}{\Pi \vdash (M_1,M_2) \in R_1 \times R_2}$

$R_1 \times R_2 \trianglelefteq R_1 \times R_2$ Reflexivity (5.8.11)

Case: $\dfrac{\Pi \vdash (M_1,M_2) \in R' \qquad R' \trianglelefteq R}{\Pi \vdash (M_1,M_2) \in R}$

$\Pi \vdash M_1 \in S_1$ and $\Pi \vdash M_2 \in S_2$
with $S_1 \times S_2 \trianglelefteq R'$ Ind. Hyp

$S_1 \times S_2 \trianglelefteq R$ Transitivity (5.8.11)

Case: $\dfrac{\Pi \vdash (V_1,V_2) \in R_1 \qquad \Pi \vdash (V_1,V_2) \in R_2}{\Pi \vdash (V_1,V_2) \in R_1 \& R_2}$

$\Pi \vdash V_1 \in S_{11}$ and $\Pi \vdash V_2 \in S_{12}$ with $S_{11} \times S_{12} \trianglelefteq R_1$	Ind. Hyp
$\Pi \vdash V_1 \in S_{21}$ and $\Pi \vdash V_2 \in S_{22}$ with $S_{21} \times S_{22} \trianglelefteq R_2$	Ind. Hyp
$\Pi \vdash V_1 \in S_{11} \& S_{21}$	&-Intro.
$\Pi \vdash V_2 \in S_{12} \& S_{22}$	&-Intro.
$S_{11} \& S_{12} \trianglelefteq S_{11}$	& is g.l.b (5.8.11)
$S_{11} \& S_{12} \trianglelefteq S_{12}$	& is g.l.b (5.8.11)
$(S_{11} \& S_{21}) \times (S_{12} \& S_{22}) \trianglelefteq S_{11} \times S_{12}$	Rules for $\times$
$(S_{11} \& S_{21}) \times (S_{12} \& S_{22}) \trianglelefteq R_1$	Transitivity (5.8.11)
$(S_{11} \& S_{21}) \times (S_{12} \& S_{22}) \trianglelefteq R_2$	As above, dual to R_1
$(S_{11} \& S_{21}) \times (S_{12} \& S_{22}) \trianglelefteq R_1 \& R_2$	& is g.l.b (5.8.11)

Case: $\dfrac{}{\Pi \vdash (V_1,V_2) \in \top^{A_1 \times A_2}}$

$\Pi \vdash V_1 \in \top^{A_1}$	Rule for $\top$
$\Pi \vdash V_2 \in \top^{A_2}$	Rule for $\top$
$\top^{A_1} \times \top^{A_1} \trianglelefteq \top^{A_1 \times A_2}$	$\top$ is maximum (5.8.11)

(c) If $\Pi \vdash \mathbf{case}\, M\, \mathbf{of}\, \Omega \in R$
then $\Pi \vdash M \in S$ and $\Pi \vdash S \bowtie \Omega \in R'$ for some $R' \trianglelefteq R$ and S.

Case: $\dfrac{\Pi \vdash M \in S \qquad \Pi \vdash S \bowtie \Omega \in R}{\Pi \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in R}$

$R \trianglelefteq R$	Reflexivity (5.8.11)

Case: $\dfrac{\Pi \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in R' \qquad R' \trianglelefteq R}{\Pi \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in R}$

$\Pi \vdash M \in S$ and $\Pi \vdash S \bowtie \Omega \in R''$ for some $R'' \trianglelefteq R'$ and S	Ind. Hyp.
$R'' \trianglelefteq R$	Transitivity (5.8.11)

3. (a) If $\Delta, P{\in}Z_1 \sqcup Z_2 \vdash J$ then $\Delta, P{\in}Z_1 \vdash J$ and $\Delta, P{\in}Z_2 \vdash J$.

The only interesting cases are the first two below, for the rule for $\sqcup$ and left subsumption since they are the only ones where the assumption $P{\in}Z_1 \sqcup Z_2$ is replaced by another assumption. Only the second one is truly interesting: the first

is immediate.

Case: $\dfrac{\Delta, P \in Z_1 \vdash M \in R \quad \Delta, P \in Z_2 \vdash M \in R}{\Delta, P \in Z_1 \sqcup Z_2 \vdash M \in R}$

Immediate.

Case: $\dfrac{Z_1 \sqcup Z_2 \preceq Y \qquad \Delta, x \in Y \vdash M \in R}{\Delta, x \in Z_1 \sqcup Z_2 \vdash M \in R}$

$Y = \sqcup\{R \mid R \text{ in } Y_1 \text{ or } Y_2\}$ with $Z_1 \preceq Y_1, Z_2 \preceq Y_2$	
$Y = ((\bot \sqcup S_1) \sqcup \ldots) \sqcup S_n$ where $\{S_1, \ldots S_n\} = \{S_i \mid S_i \text{ in } Y_1 \text{ or } Y_2\}$	Inv. on $Z_1 \sqcup Z_2 \preceq Y$
$\Delta, x \in S_i \vdash M \in R$ for each S_i	Ind. Hyp., repeatedly
$\Delta, x \in Y_1 \vdash M \in R$	Rule for $\sqcup$, repeatedly
$\Delta, x \in Y_2 \vdash M \in R$	Rule for $\sqcup$, repeatedly
$\Delta, x \in Z_1 \vdash M \in R$	Pattern subsumption
$\Delta, x \in Z_2 \vdash M \in R$	Pattern subsumption

For this part we additionally show some of the cases that simply rebuild the derivation, although we skip these in the later parts, since they are completely straightforward.

Case: $\dfrac{x \in R \text{ in } (\Delta, P \in Z_1 \sqcup Z_2)}{\Delta, P \in Z_1 \sqcup Z_2 \vdash x \in R}$

$x \in R$ in Δ	$R \neq Z_1 \sqcup Z_2$
$\Delta, P \in Z_1 \vdash x \in R$	Rule for x
$\Delta, P \in Z_2 \vdash x \in R$	Rule for x

Case: $\dfrac{\Delta, P \in Z_1 \sqcup Z_2, x \in R \vdash M \in S}{\Delta, P \in Z_1 \sqcup Z_2 \vdash \lambda x.M \in R \to S}$

$\Delta, P \in Z_1, x \in R \vdash M \in S$ and $\Delta, P \in Z_2, x \in R \vdash M \in S$	Ind. Hyp.
$\Delta, P \in Z_1 \vdash \lambda x.M \in R \to S$	Rule for λ
$\Delta, P \in Z_2 \vdash \lambda x.M \in R \to S$	Rule for λ

Case: $\dfrac{\Delta, P{\in}Z_1 \sqcup Z_2 \vdash M \in R \qquad \Delta, P{\in}Z_1 \sqcup Z_2 \vdash R \bowtie \Omega \in S}{\Delta, P{\in}Z_1 \sqcup Z_2 \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S}$

$\Delta, P{\in}Z_1, x{\in}R \vdash M \in S$ and
$\Delta, P{\in}Z_2, x{\in}R \vdash M \in S$ Ind. Hyp.

$\Delta, P{\in}Z_1 \vdash R \bowtie \Omega \in S$ and
$\Delta, P{\in}Z_2 \vdash R \bowtie \Omega \in S$ Ind. Hyp.

$\Delta, P{\in}Z_1 \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S$ Rule for **case**
$\Delta, P{\in}Z_2 \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S$ Rule for **case**

Case: $\dfrac{\Delta, P{\in}Z_1 \sqcup Z_2, P'{\in}Z \vdash M{\in}R \qquad \Delta, P{\in}Z_1 \sqcup Z_2 \vdash (Z \backslash P') \bowtie \Omega \in R}{\Delta, P{\in}Z_1 \sqcup Z_2 \vdash Z \bowtie (P' \Rightarrow M \mid \Omega) \in R}$

$\Delta, P{\in}Z_1, P'{\in}Z \vdash M \in R$ and
$\Delta, P{\in}Z_2, P'{\in}Z \vdash M \in R$ Ind. Hyp.

$\Delta, P{\in}Z_1 \vdash (Z \backslash P') \bowtie \Omega \in R$ and
$\Delta, P{\in}Z_2 \vdash (Z \backslash P') \bowtie \Omega \in R$ Ind. Hyp.

$\Delta, P{\in}Z_1 \vdash Z \bowtie (P' \Rightarrow M \mid \Omega) \in R$ Rule for $P \Rightarrow M \mid \Omega$
$\Delta, P{\in}Z_1 \vdash Z \bowtie (P' \Rightarrow M \mid \Omega) \in R$ Rule for $P \Rightarrow M \mid \Omega$

Case: *The remaining right rules.*

Each similarly rebuilds the derivation after applying the induction hypothesis to the premises.

Case: $\dfrac{\Delta, P_1{\in}Z_1', P_2{\in}Z_2', P{\in}Z_1 \sqcup Z_2 \vdash M{\in}R}{\Delta, (P_1,P_2){\in}(Z_1',Z_2'), P{\in}Z_1 \sqcup Z_2 \vdash M{\in}R}$

$\Delta, P_1{\in}Z_1', P_2{\in}Z_2', P{\in}Z_1 \vdash M{\in}R$ and
$\Delta, P_1{\in}Z_1', P_2{\in}Z_2', P{\in}Z_2 \vdash M{\in}R$ Ind. Hyp.

$\Delta, (P_1,P_2){\in}(Z_1',Z_2'), P{\in}Z_1 \vdash M \in R$ Rule for (Z_1,Z_2)
$\Delta, (P_1,P_2){\in}(Z_1',Z_2'), P{\in}Z_2 \vdash M \in R$ Rule for (Z_1,Z_2)

Case: $\dfrac{\Delta, P'{\in}Z_1', P{\in}Z_1 \sqcup Z_2 \vdash M{\in}R \qquad \Delta, P'{\in}Z_2', P{\in}Z_1 \sqcup Z_2 \vdash M{\in}R}{\Delta, P'{\in}Z_1' \sqcup Z_2', P{\in}Z_1 \sqcup Z_2 \vdash M{\in}R}$

$\Delta, P'{\in}Z_1', P{\in}Z_1 \vdash M \in R$ and
$\Delta, P'{\in}Z_1', P{\in}Z_2 \vdash M \in R$ Ind. Hyp.

$\Delta, P'{\in}Z_2', P{\in}Z_1 \vdash M \in R$ and
$\Delta, P'{\in}Z_2', P{\in}Z_2 \vdash M \in R$ Ind. Hyp.

$\Delta, P'{\in}Z_1' \sqcup Z_2', P{\in}Z_1 \vdash M \in R$ Rule for $\sqcup$

$\Delta, P'{\in}Z_1' \sqcup Z_2', P{\in}Z_2 \vdash M \in R$ Rule for $\sqcup$

Case: *The remaining left rules.*

Each similarly rebuilds the derivation after applying the induction hypothesis to the premises.

(b) If $\Delta, (P_1{,}P_2){\in}S \vdash J$ then $S \stackrel{\rightrightarrows}{\simeq} S_1 \otimes S_2$ and $\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash J$.

Case: $\dfrac{\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash M{\in}R}{\Delta, (P_1{,}P_2){\in}S_1{\times}S_2 \vdash M{\in}R}$

$S_1 \times S_2 \stackrel{\rightrightarrows}{\simeq} S_1 \otimes S_2$ Rule for $\times$

Case: $\dfrac{S \preceq Y \qquad \Delta, (P_1{,}P_2) \in Y \vdash M{\in}R}{\Delta, (P_1{,}P_2){\in}S \vdash M{\in}R}$

$\Delta, (P_1{,}P_2){\in}S \sqsubset \Gamma$ with $\Gamma \vdash M{\in}R$ Well-formedness

$\Gamma = \Gamma', (P_1{,}P_2){\in}A$ with $S{\sqsubset}A$ Inv. on $\Delta, (P_1{,}P_2){\in}S \sqsubset \Gamma$

$A = A_1 \times A_2$ Inv. on type deriv.

$S \stackrel{\rightrightarrows}{\simeq} S_1 \otimes S_2$ Lemma 5.6.8

Then, we have two subcases for the derivation of $S \preceq Y$.

Subcase: $\dfrac{S \trianglelefteq S'}{S \trianglelefteq S'}$

$S' \stackrel{\rightrightarrows}{\cong} S'_1 \otimes S'_2$ and
$\Delta, P_1{\in}S'_1, P_2{\in}S'_2 \vdash M{\in}R$ — Ind. Hyp.

$S' \trianglelefteq S'_1 \times S'_2$ — Lemma 5.6.8

$S \trianglelefteq S'_1 \times S'_2$ — Transitivity (5.8.11)

$S_1 \times S_2 \trianglelefteq S'_1 \times S'_2$ — Transitivity (5.8.11)

($S_1 \trianglelefteq S'_1$ and $S_2 \trianglelefteq S'_2$)
or $S_1 \trianglelefteq \cdot$ or $S_2 \trianglelefteq \cdot$ — Prod. Inv. Lem. 6.6.6

If $S_1 \trianglelefteq S'_1$ and $S_2 \trianglelefteq S'_2$:

$\Delta, P_1{\in}S_1, P_2{\in}S'_2 \vdash M{\in}R$ — Left Subsumption

$\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash M{\in}R$ — Left Subsumption

If $S_1 \trianglelefteq \cdot$ then:

$S_1 \trianglelefteq \bot$

$\Delta, P_1{\in}\bot, P_2{\in}S_2 \vdash M{\in}R$ — Rule for $\bot$

$\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash M{\in}R$ — Left Subsumption

If $S_2 \trianglelefteq \cdot$ then:
Dual to that for $S_1 \trianglelefteq \cdot$

Subcase: $\dfrac{S \trianglelefteq \cdot}{S \trianglelefteq \bot}$

$S_1 \times S_2 \trianglelefteq \cdot$ — Lemmas 5.6.8, 5.8.10

$S_1 \trianglelefteq \cdot$ or $S_2 \trianglelefteq \cdot$

If $S_1 \trianglelefteq \cdot$ then:

$S_1 \trianglelefteq \bot$

$\Delta, P_1{\in}\bot, P_2{\in}S_2 \vdash M{\in}R$ — Rule for $\bot$

$\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash M{\in}R$ — Left Subsumption

If $S_2 \trianglelefteq \cdot$ then:
Dual to that for $S_1 \trianglelefteq \cdot$

(c) If $\Delta, (P_1{,}P_2){\in}(Z_1{,}Z_2) \vdash J$ then $\Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash J$.

There is no subsumption case for this part: neither subsumption rule matches the conclusion. Hence, the following is only the case which does not simply rebuild the derivation.

Case: $\dfrac{\Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash M{\in}R}{\Delta, (P_1, P_2) \in (Z_1{,}Z_2) \vdash M{\in}R}$

Immediate.

(d) If $\Delta, ()\in() \vdash J$ then $\Delta \vdash J$.

Again, neither subsumption rule matches the conclusion for this part. Hence, the following is only the case which does not simply rebuild the derivation.

Case: $\dfrac{\Delta \vdash M\in R}{\Delta, () \in () \vdash M\in R}$

Immediate.

(e) If $\Delta, ()\in S \vdash J$ then $\Delta \vdash J$.

Case: $\dfrac{\Delta \vdash M\in R}{\Delta, () \in S \vdash M\in R}$

Immediate.

Case: $\dfrac{S \preceq Y \qquad \Delta, () \in Y \vdash M\in R}{\Delta, () \in S \vdash M\in R}$

$(Y = S'$ and $S \trianglelefteq S')$ or $(Y = \bot$ and $S \trianglelefteq \cdot)$ Inversion

If $Y = S'$ and $S \trianglelefteq S'$

$\Delta, () \in S' \vdash M\in R$ Substituting $Y = S'$

$\Delta \vdash M\in R$ Ind. Hyp.

If $Y = \bot$ and $S \trianglelefteq \cdot$ then:

$S \sqsubset 1$ Well-formedness

$[\![S]\!] = \{()\}$ Lemma 5.6.7

$[\![\cdot]\!] = \{\}$ Def. $[\![\cdot]\!]$

$S \not\trianglelefteq \cdot$ Soundness of $\trianglelefteq$ (5.8.2)

Contradicts $S \trianglelefteq \cdot$, case cannot occur.

(f) If $\Delta, cP\in cZ \vdash J$ then $\Delta, P\in Z \vdash J$.

Again, neither subsumption rule matches the conclusion for this part. Hence, the following is only the case which does not simply rebuild the derivation.

Case: $\dfrac{\Delta, P\in Z \vdash M\in R}{\Delta, cP\in cZ \vdash M\in R}$

Immediate.

(g) If $\Delta, cP\in\rho \vdash J$ then $\Delta, cP\in\mathsf{inv}(\rho) \vdash J$.

Case: $\dfrac{\Delta, cP\in\mathsf{inv}(\rho) \vdash M\in R}{\Delta, cP\in\rho \vdash M\in R}$

Immediate.

Case: $\dfrac{\rho \preceq Y \qquad \Delta, cP{\in}Y \vdash M{\in}R}{\Delta, cP{\in}\rho \vdash M{\in}R}$

Subcase: $\dfrac{\rho \trianglelefteq \rho'}{\rho \preceq \rho'}$

$\Delta, cP{\in}\rho' \vdash M{\in}R$	Substituting $Y = \rho'$
$\Delta, cP{\in}\mathsf{inv}(\rho') \vdash M{\in}R$	Ind. Hyp.
For each $c'S'$ in $\mathsf{inv}(\rho')$:	
$\Delta, cP{\in}c'S' \vdash M{\in}R$	Part 3a, repeatedly
Now, for each cS in $\mathsf{inv}(\rho)$:	
cS' in $\mathsf{inv}(\rho')$ with $S \trianglelefteq S'$	Mono. of inv (6.6.10)
$\Delta, cP{\in}cS' \vdash M{\in}R$	Instance of above
$\Delta, P{\in}S' \vdash M{\in}R$	Part 3f
$\Delta, P{\in}S \vdash M{\in}R$	Left Subsumption Rule
$\Delta, cP{\in}cS \vdash M{\in}R$	Rule for $cP{\in}cS$
And, for each $c'S$ in $\mathsf{inv}(\rho)$ with $c' \neq c$:	
$\Delta, cP{\in}c'S \vdash M{\in}R$	Rule for $c' \neq c$
So: $\Delta, cP{\in}\mathsf{inv}(\rho) \vdash M{\in}R$	Rule for $\sqcup$, repeatedly

Subcase: $\dfrac{\rho \trianglelefteq \cdot}{\rho \preceq \bot}$

$\&\{r \mid \rho \trianglelefteq r\} \trianglelefteq \rho$	$\&$ is g.l.b. (5.8.11)
$\&\{r \mid \rho \trianglelefteq r\} \trianglelefteq \cdot$	Transitivity (5.8.11)
$\mathsf{body}(\&\{r \mid \rho \trianglelefteq r\}) \trianglelefteq \cdot$	Lemma 5.6.2
$\mathsf{inv}(\rho) \trianglelefteq \cdot$	Def. inv
Now, for each cS in $\mathsf{inv}(\rho)$:	
$S \trianglelefteq \cdot$	Inversion on $\mathsf{inv}(\rho) \trianglelefteq \cdot$
$S \preceq \bot$	Rule for $S \preceq \bot$
$\Delta, P{\in}\bot \vdash M{\in}R$	Rule for $\bot$
$\Delta, P{\in}S \vdash M{\in}R$	Left Subsumption
$\Delta, cP{\in}cS \vdash M{\in}R$	Rule for $cP{\in}cS$
And, for each $c'S$ in $\mathsf{inv}(\rho)$ with $c' \neq c$:	
$\Delta, cP{\in}c'S \vdash M{\in}R$	Rule for $c' \neq c$
So: $\Delta, cP{\in}\mathsf{inv}(\rho) \vdash M{\in}R$	Rule for $\sqcup$, repeatedly

(h) If $\Delta, x{\in}Z \vdash J$
then $Z \overset{\Rightarrow}{\preceq} Y$ and for each S in Y we have $\Delta, x{\in}S \vdash J$.

For this part the subsumption case coincides with the case for the rule being inverted. However, we have an additional interesting case: that for uses of the variable x when Z is a sort.

Case: $\dfrac{Z \preceq Y' \qquad \Delta, x{\in}Y' \vdash M{\in}R}{\Delta, x{\in}Z \vdash M{\in}R}$

$Z \overset{\Rightarrow}{\preceq} Y$ with S in Y implies $S \trianglelefteq S'$ for some S' in Y' — Compl. $\overset{\Rightarrow}{\preceq}$ (6.7.8)

$\Delta, x{\in}S' \vdash M{\in}R$ for each S' in Y' — Part 3a, repeatedly

$\Delta, x{\in}S \vdash M{\in}R$ for each S in Y — Left subsumption rule

Case: $\dfrac{x{\in}R \text{ in } \Delta, x{\in}R}{\Delta, x{\in}R \vdash x{\in}R}$

$R \preceq R$ — Rule for $\preceq$

$R \overset{\Rightarrow}{\preceq} Y$ with S in Y implies $S \trianglelefteq R$ — Compl. $\overset{\Rightarrow}{\preceq}$ (6.7.8)

$\Delta, x{\in}S \vdash x{\in}R$ for each S in Y — Left subsumption rule

□

In the completeness proof we generally need to account for the possibility that the algorithm may use stronger sorts than are assigned by a declarative derivation. In the case of the judgment $\Delta \vdash Z \bowtie \Omega \in S$ this requires a suitable notion of inclusion for the pattern sort Z. The judgment $Z \preceq Y$ is not a suitable for this purpose: it is designed to distribute unions outwards so that Y is always a sort union. We thus introduce a new judgment $Z \subseteq Z'$ which is designed to satisfy the following properties: it generalizes subsorting $R \trianglelefteq S$ and it is preserved by pattern subtraction.

$Z \subseteq Z'$ Z is compositionally included in Z'
(Where $Z \sqsubset A$ and $Z' \sqsubset A$.)

$$\frac{R \trianglelefteq S}{R \subseteq S} \qquad \frac{R \trianglelefteq \cdot}{R \subseteq Z} \qquad \frac{Z \subseteq Z'}{cZ \subseteq cZ'} \qquad \frac{Z \subseteq \bot}{cZ \subseteq Z'} \qquad \frac{}{() \subseteq ()}$$

$$\frac{Z_1 \subseteq Z_1' \qquad Z_2 \subseteq Z_2'}{(Z_1,Z_2) \subseteq (Z_1',Z_2')} \qquad \frac{Z_1 \subseteq \bot}{(Z_1,Z_2) \subseteq Z'} \qquad \frac{Z_2 \subseteq \bot}{(Z_1,Z_2) \subseteq Z'}$$

$$\frac{}{\bot \subseteq Z'} \qquad \frac{Z_1 \subseteq Z' \qquad Z_2 \subseteq Z'}{Z_1 \sqcup Z_2 \subseteq Z'} \qquad \frac{Z \subseteq Z_1'}{Z \subseteq Z_1' \sqcup Z_2'} \qquad \frac{Z \subseteq Z_2'}{Z \subseteq Z_1' \sqcup Z_2'}$$

Lemma 6.7.10 (Preservation of $\subseteq$)
If $Z \subseteq Z'$ then $Z \backslash P \subseteq Z' \backslash P$.

Proof: By induction on the definition of $Z\backslash P$ and the derivation of $Z \subseteq Z'$, lexicographically.

We first treat the four rules for $Z \subseteq Z'$ involving $\sqcup$ and $\bot$. We then consider the remaining cases following the definition of $Z'\backslash P$, using inversion from Z to determine which of the eight remaining rules matches the conclusion $Z \subseteq Z'$.

Case: $\dfrac{}{\bot \subseteq Z'}$. Then $\bot\backslash P = \bot$ and $\bot \subseteq Z'\backslash P$ (rule for $\bot$).

Case: $\dfrac{Z_1 \subseteq Z' \qquad Z_2 \subseteq Z'}{Z_1 \sqcup Z_2 \subseteq Z'}$

$Z_1\backslash P \subseteq Z'\backslash P$ and $Z_2\backslash P \subseteq Z'\backslash P$	Ind. Hyp.
$(Z_1\backslash P) \sqcup (Z_2\backslash P) \subseteq Z'\backslash P$	Rule for $\sqcup$
$(Z_1 \sqcup Z_2) \setminus P \subseteq Z'\backslash P$	Def. $\setminus$

Case: $\dfrac{Z \subseteq Z_1'}{Z \subseteq Z_1' \sqcup Z_2'}$

$Z\backslash P \subseteq Z_1'\backslash P$	Ind. Hyp.
$Z\backslash P \subseteq (Z_1'\backslash P) \sqcup (Z_2'\backslash P)$	Rule for $\sqcup$
$Z\backslash P \subseteq (Z_1' \sqcup Z_2') \setminus P$	Def. $\setminus$

Case: $\dfrac{Z \subseteq Z_2'}{Z \subseteq Z_1' \sqcup Z_2'}$. Dual to the previous case.

Case: $Z\backslash x = \bot$. Then $\bot \subseteq Z'\backslash x$ (rule for $\bot$).

Case: $Z \setminus () = \bot$. Then $\bot \subseteq Z'\backslash()$ (rule for $\bot$).

Case: $(Z_1, Z_2) \setminus (P_1, P_2) = ((Z_1\backslash P_1), Z_2) \sqcup (Z_1, (Z_2\backslash P_2))$

By inversion on $Z \subseteq Z'$ we have one of the following subcases.

Subcase: $Z' = (Z_1', Z_2')$ and $Z_1 \subseteq Z_1'$ and $Z_2 \subseteq Z_2'$

$(Z_1', Z_2') \setminus (P_1, P_2) = ((Z_1'\backslash P_1), Z_2') \sqcup (Z_1', (Z_2'\backslash P_1))$	Def. $\setminus$
$Z_1\backslash P \subseteq Z_1'\backslash P$	Ind. Hyp.
$((Z_1\backslash P_1), Z_2) \subseteq ((Z_1'\backslash P_1), Z_2')$	Rule for pairs
$((Z_1\backslash P_1), Z_2) \subseteq (Z_1', Z_2') \setminus (P_1, P_2)$	Right rule for $\sqcup$
$(Z_1, (Z_2\backslash P_2)) \subseteq (Z_1', Z_2') \setminus (P_1, P_2)$	Similarly, dual to above
$(Z_1, Z_2) \setminus (P_1, P_2) \subseteq (Z_1', Z_2') \setminus (P_1, P_2)$	Left rule for $\sqcup$

Subcase: $Z_1 \subseteq \bot$

$Z_1\backslash P_1 \subseteq \bot\backslash P_1$	Ind. Hyp.
$Z_1\backslash P_1 \subseteq \bot$	Def. $\setminus$
$((Z_1\backslash P_1), Z_2) \subseteq Z'\backslash(P_1, P_2)$	Rule for first part empty

$(Z_1,(Z_2\backslash P_2)) \subseteq Z'\backslash(P_1,P_2)$ — Rule for first part empty

$(Z_1,Z_2)\backslash(P_1,P_2) \subseteq Z'\backslash(P_1,P_2)$ — Left rule for $\sqcup$

Subcase: $Z_2 \subseteq \bot$

Dual to that for $Z_1 \subseteq \bot$.

Case: $R\backslash(P_1,P_2) = ((R_1\backslash P_1),R_2) \sqcup (R_1,(R_2\backslash P_2))$ with $R \stackrel{\rightrightarrows}{\cong} R_1 \otimes R_2$

By inversion on $Z \subseteq Z'$ we have one of the following subcases.

Subcase: $R \trianglelefteq \cdot$

$R_1 \times R_2 \trianglelefteq \cdot$ — Transitivity (5.8.11)

$R_1 \trianglelefteq \cdot$ or $R_2 \trianglelefteq \cdot$ — Inversion

If $R_1 \trianglelefteq \cdot$ then:

$R_1 \subseteq \bot$ — Rule for empty sort

$R_1\backslash P_1 \subseteq \bot\backslash P_1$ — Ind. Hyp.

$R_1\backslash P_1 \subseteq \bot$ — Def. $\backslash$

$((R_1\backslash P_1),R_2) \subseteq Z'\backslash(P_1,P_2)$ — Rule for first part empty

$(R_1,(R_2\backslash P_2)) \subseteq Z'\backslash(P_1,P_2)$ — Rule for first part empty

$R\backslash(P_1,P_2) \subseteq Z'\backslash(P_1,P_2)$ — Left rule for $\sqcup$

If $R_2 \trianglelefteq \cdot$ then:

Dual to that for $R_1 \trianglelefteq \cdot$.

Subcase: $Z' = R'$ and $R \trianglelefteq R'$

$R' \stackrel{\rightrightarrows}{\cong} R'_1 \otimes R'_2$ — Lemma 5.6.8

$R'\backslash(P_1,P_2) = ((R'_1\backslash P_1),R'_2) \sqcup (R'_1,(R'_2\backslash P_1))$ — Def. $\backslash$

$R_1 \times R_2 \trianglelefteq R'_1 \times R'_2$ — Transitivity (5.8.11)

$(R_1 \trianglelefteq R'_1$ and $R_2 \trianglelefteq R'_2)$
or $R_1 \trianglelefteq \cdot$ or $R_2 \trianglelefteq \cdot$ — Prod. Inv. Lemma 6.6.6

If $R_1 \trianglelefteq R'_1$ and $R_2 \trianglelefteq R'_2$ then:

$R_1\backslash P_1 \subseteq R'_1\backslash P_1$ and $R_2\backslash P_2 \subseteq R'_2\backslash P_2$ — Ind. Hyp.

$((R_1\backslash P_1),R_2) \subseteq ((R'_1\backslash P_1),R'_2)$ — Rule for pairs

$((R_1\backslash P_1),R_2) \subseteq R'\backslash(P_1,P_2)$ — Right rule for $\sqcup$

$(R_1,(R_2\backslash P_2)) \subseteq R'\backslash(P_1,P_2)$ — Similarly, dual to above

$R\backslash(P_1,P_2) \subseteq R'\backslash(P_1,P_2)$ — Left rule for $\sqcup$

If $R_1 \trianglelefteq \cdot$ then:

As for the case for $R_1 \trianglelefteq \cdot$ in the previous subcase.

If $R_2 \trianglelefteq \cdot$ then:

As for the case for $R_2 \trianglelefteq \cdot$ in the previous subcase.

Case: $(c\,Z_1) \setminus (c\,P_1) = c\,(Z_1 \backslash P_1)$

By inversion on $Z \subseteq Z'$ we have one of the following subcases.

Subcase: $Z' = c\,Z_1'$ with $Z_1 \subseteq Z_1'$

$(c\,Z_1') \setminus (c\,P_1) = c\,(Z_1' \backslash P_1)$	Def. $\setminus$
$Z_1 \backslash P_1 \subseteq Z_1' \backslash P_1$	Ind. Hyp.
$c(Z_1 \backslash P_1) \subseteq c(Z_1' \backslash P_1)$	Rule for c

Subcase: $Z_1 \subseteq \bot$

$Z_1 \backslash P_1 \subseteq \bot \backslash P_1$	Ind. Hyp.
$Z_1 \backslash P_1 \subseteq \bot$	Def. $\setminus$
$c(Z_1 \backslash P_1) \subseteq Z' \backslash (c P_1)$	Rule for c with $\bot$

Case: $(c\,Z_1) \setminus (c_2\,P_2) = c\,Z_1 \qquad$ with $c_2 \neq c$

By inversion on $Z \subseteq Z'$ we have one of the following subcases.

Subcase: $Z' = c\,Z_1'$

$(c\,Z_1') \setminus (c_2\,P_2) = c\,Z_1'$	Def. $\setminus$
$c\,Z_1 \subseteq c\,Z_1'$	Assumed

Subcase: $Z_1 \subseteq \bot$

$c\,Z_1 \subseteq Z' \backslash (c_2\,P_2)$	Rule for c with $\bot$

Case: $\rho \setminus (c\,P) = \bigsqcup \{c(R \backslash P) \mid c\,R \text{ in } \mathsf{inv}(\rho)\} \sqcup \bigsqcup \{c_1\,R_1 \mid c_1\,R_1 \text{ in } \mathsf{inv}(\rho) \text{ and } c_1 \neq c\}$

By inversion on $Z \subseteq Z'$ we have one of the following subcases.

Subcase: $\rho \trianglelefteq \cdot$

$\&\{r \mid \rho \trianglelefteq r\} \trianglelefteq \rho$		& is g.l.b. (5.8.11)
$\&\{r \mid \rho \trianglelefteq r\} \trianglelefteq \cdot$		Transitivity (5.8.11)
$\mathsf{body}(\&\{r \mid \rho \trianglelefteq r\}) \trianglelefteq \cdot$		Lemma 5.6.2
$\mathsf{inv}(\rho) \trianglelefteq \cdot$		Def. inv

Now, for each $c_1\,R$ in $\mathsf{inv}(\rho)$:

$R \trianglelefteq \cdot$		Inversion on $\mathsf{inv}(\rho) \trianglelefteq \cdot$
$R \subseteq \bot$		Rule for empty sort
$R \backslash P \subseteq \bot$	when $c_1 = c$	Ind. Hyp.
$c(R \backslash P) \subseteq Z' \backslash (cP)$	when $c_1 = c$	Rule for c with $\bot$
$c_1\,R \subseteq Z' \backslash (cP)$	when $c_1 \neq c$	Rule for c with $\bot$

So, for each $c_1\,S$ in $\rho \backslash (cP)$:

$c_1\,S \subseteq Z' \backslash (cP)$	Def. $\setminus$
$\rho \backslash (cP) \subseteq Z' \backslash (cP)$	Rule for $\sqcup$, repeatedly

Subcase: $Z' = \rho'$ and $\rho \trianglelefteq \rho'$

Then, for each $c_1\, R$ in $\mathsf{inv}(\rho)$:

$R \trianglelefteq R'$ for some $c\, R'$ in $\mathsf{inv}(\rho')$		Mono. of inv (6.6.10)
$R \subseteq R'$		Rule for sort
$R \backslash P \subseteq R' \backslash P$	when $c_1 = c$	Ind. Hyp.
$c(R \backslash P) \subseteq c(R' \backslash P)$	when $c_1 = c$	Rule for c
$c(R \backslash P) \subseteq \rho' \backslash (c P)$	when $c_1 = c$	Right $\sqcup$ rule, repeatedly
$c_1\, R \subseteq c_1 R'$	when $c_1 \neq c$	Rule for c
$c_1\, R \subseteq \rho' \backslash (c P)$	when $c_1 \neq c$	Right rule for $\sqcup$, repeatedly
$\rho \setminus (c\, P) \subseteq \rho' \backslash (c P)$		Left rule for $\sqcup$, repeatedly

□

We continue with two small lemmas. The first relates $\subseteq$ to $\preceq$ for empty sorts and the second is a generalized form of inversion for $Y \subseteq Y'$ when Y and Y' are sort unions.

Lemma 6.7.11 *If $Z \subseteq \bot$ then $Z \preceq \bot$.*

Proof: By induction on the structure of the derivation of $Z \subseteq \bot$. We have one case for each rule which has a conclusion $Z \subseteq Z'$ with Z' not required to be of a particular form.

Case: $\dfrac{R \trianglelefteq \cdot}{R \subseteq \bot}$ Then $R \preceq \bot$ (rule for $\bot$).

Case: $\dfrac{Z \subseteq \bot}{c Z \subseteq \bot}$ Then $Z \preceq \bot$ (ind. hyp.)
so $c Z \preceq \bot$ (rule for c).

Case: $\dfrac{Z_1 \subseteq \bot}{(Z_1, Z_2) \subseteq \bot}$ Then $Z_1 \preceq \bot$ (ind. hyp.)
so $(Z_1, Z_2) \preceq \bot$ (rule for pairs).

Case: $\dfrac{Z_2 \subseteq \bot}{(Z_1, Z_2) \subseteq \bot}$ Dual to the previous case.

Case: $\dfrac{}{\bot \subseteq \bot}$ Then $\bot \preceq \bot$ (rule for $\bot$).

Case: $\dfrac{Z_1 \subseteq \bot \quad Z_2 \subseteq \bot}{Z_1 \sqcup Z_2 \subseteq \bot}$ Then $Z_1 \preceq \bot$ and $Z_2 \preceq \bot$ (ind. hyp.)
so $Z_1 \sqcup Z_2 \preceq \bot$ (rule for $\sqcup$).

□

Lemma 6.7.12
If $Y \subseteq Y'$ and R in Y then $R \trianglelefteq \cdot$ or $R \trianglelefteq S$ for some S in Y'.

Proof: By induction on structure of the derivation of $Y \subseteq Y'$.

Case: $\dfrac{R \trianglelefteq R'}{R \subseteq R'}$ Immediate.

Case: $\dfrac{R \trianglelefteq \cdot}{R \subseteq Z'}$ Immediate.

Case: $\dfrac{Y_1 \subseteq Y' \qquad Y_2 \subseteq Y'}{Y_1 \sqcup Y_2 \subseteq Y'}$

R in Y_1 or R in Y_2

$R \trianglelefteq \cdot$ or $R \trianglelefteq S$ for some S in Y' Ind. hyp. in each case

Case: $\dfrac{Y \subseteq Y_1'}{Y \subseteq Y_1' \sqcup Y_2'}$

$R \trianglelefteq \cdot$ or $R \trianglelefteq S$ for some S in Y_1' Ind. hyp.

$R \trianglelefteq \cdot$ or $R \trianglelefteq S$ for some S in $Y_1' \sqcup Y_2'$ Def. "in"

Case: $\dfrac{Y \subseteq Y_2'}{Y \subseteq Y_1' \sqcup Y_2'}$ Dual to the previous case.

□

Next, we prove a lemma which relates the judgment $Z \subseteq Z'$ to the sort union inclusion judgment $Z \preceq Y$, making use of the preceding two lemmas.

Lemma 6.7.13
If $Z \subseteq Z'$ and $Z' \preceq Y'$ then there is some Y such that the following hold.

- *$Z \preceq Y$ and $Y \subseteq Y'$.*
- *$R \not\trianglelefteq \cdot$ for each R in Y.*

Proof: By induction on the structure of the derivations of $Z' \preceq Y'$ and $Z \subseteq Z'$, lexicographically.

Case: $\dfrac{R \trianglelefteq R'}{R \subseteq R'}$ and $\dfrac{R' \trianglelefteq S'}{R' \preceq S'}$

Subcase: $R' \not\trianglelefteq \cdot$

Then $R \preceq R'$ (rule for sorts) and $R' \subseteq S'$ (rule for sorts).

Subcase: $R' \trianglelefteq \cdot$

Then $R \trianglelefteq \cdot$ (trans.), so $R \preceq \bot$ (rule) and $\bot \subseteq S'$ (rule).

Case: $\dfrac{R \trianglelefteq R'}{R \subseteq R'}$ and $\dfrac{R' \trianglelefteq \cdot}{R' \preceq \bot}$

Then $R \trianglelefteq \cdot$ (trans.), so $R \preceq \bot$ (rule) and $\bot \subseteq \bot$ (rule).

Case: $\dfrac{R \trianglelefteq \cdot}{R \subseteq Z'}$ and $Z' \preceq Y'$

Then $R \preceq \bot$ (rule for empty sort) and $\bot \subseteq Y'$ (rule for empty sort).

Case: $\dfrac{Z \subseteq Z'}{c\,Z \subseteq c\,Z'}$ and $\dfrac{Z' \preceq R_1 \sqcup \ldots \sqcup R_n \qquad c\,R_1 \rightarrowtail \rho_1 \;\; \ldots \;\; c\,R_n \rightarrowtail \rho_n}{c\,Z' \preceq \rho_1 \sqcup \ldots \sqcup \rho_n}$

$Z \preceq S_1 \sqcup \ldots \sqcup S_m$ and $S_1 \sqcup \ldots \sqcup S_m \subseteq R_1 \sqcup \ldots \sqcup R_n$
with not $S_i \trianglelefteq \cdot$ for each S_i — Ind. hyp.

For each S_i:

$S_i \trianglelefteq \cdot$ or $S_i \trianglelefteq R_j$ for some R_j — Lemma 6.7.12
not $S_i \trianglelefteq \cdot$ in Y — Above
$S_i \trianglelefteq R_j$ for some R_j — Other case is impossible

Let $\hat{Y}$ be such that for each S_i, $\hat{Y}$ contains ρ_j (with j chosen as above).

Then, for each ρ_j and corresponding S_i:

$c\,R$ in $\mathsf{body}(\rho_j)$ with $R_j \trianglelefteq R$ — Inv. on $\rightarrowtail$
$S_i \trianglelefteq R$ — Transitivity
$c\,S_i \rightarrowtail \rho_j$ — Rule for $\rightarrowtail$
not $R \trianglelefteq \cdot$ — Since not $S_i \trianglelefteq \cdot$, transitivity
not $\mathsf{body}(\rho_j) \trianglelefteq \cdot$ — Requires $R \trianglelefteq \cdot$, by inversion
not $\rho_j \trianglelefteq \cdot$ — Lemmas 5.6.2, 5.8.10

So:

$c\,Z \preceq \hat{Y}$ — Rule for c
not $\rho_j \trianglelefteq \cdot$ for each ρ_j in $\hat{Y}$ — Above
$\rho_j \subseteq \rho_1 \sqcup \ldots \sqcup \rho_n$ for each ρ_j — Right rule for $\sqcup$, repeatedly
$\hat{Y} \subseteq \rho_1 \sqcup \ldots \sqcup \rho_n$ — Left rule for $\sqcup$, repeatedly

Case: $\dfrac{Z \subseteq \bot}{c\,Z \subseteq Z'}$ and $Z' \preceq Y'$

Then $Z \preceq \bot$ (lemma 6.7.11) and $\bot \subseteq Y'$ (rule for $\bot$).

Case: $\dfrac{}{() \subseteq ()}$ and $\dfrac{}{() \preceq 1}$

Then $() \preceq 1$ (rule) and $1 \subseteq 1$ (rule for sorts, via reflexivity).

Case: $\dfrac{Z_1 \subseteq Z_1' \qquad Z_2 \subseteq Z_2'}{(Z_1,Z_2) \subseteq (Z_1',Z_2')}$ and $\dfrac{Z_1' \preceq Y_1' \qquad Z_2' \preceq Y_2'}{(Z_1',Z_2') \preceq \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1', R_2 \text{ in } Y_2'\}}$

$Z_1 \preceq Y_1$ and $Y_1 \subseteq Y_1'$ for some Y_1 — Ind. hyp.
$Z_2 \preceq Y_2$ and $Y_2 \subseteq Y_1'$ for some Y_2 — Ind. hyp.
$(Z_1,Z_2) \preceq \sqcup\{S_1 \times S_2 \mid S_1 \text{ in } Y_1, S_2 \text{ in } Y_2\}$ — Rule for pairs

If S_1 in Y_1 and S_2 in Y_2 then:

$S_1 \trianglelefteq \cdot$ or $S_1 \trianglelefteq R_1$ for some R_1 in Y_1' — Lemma 6.7.12

$S_2 \trianglelefteq \cdot$ or $S_2 \trianglelefteq R_2$ for some R_2 in Y_2' — Lemma 6.7.12

$S_1 \times S_2 \trianglelefteq R_1 \times R_2$ — Rules for $\times$, in each case

$S_1 \times S_2 \subseteq R_1 \times R_2$ — Rule for sorts

$S_1 \times S_2 \subseteq \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1', R_2 \text{ in } Y_2'\}$ — Right $\sqcup$ rule, rep.

$\sqcup\{S_1 \times S_2 \mid S_1 \text{ in } Y_1, S_2 \text{ in } Y_2\} \subseteq \sqcup\{R_1 \times R_2 \mid R_1 \text{ in } Y_1', R_2 \text{ in } Y_2'\}$ — Left $\sqcup$ rule, rep.

Case: $\dfrac{Z_1 \subseteq \bot}{(Z_1,Z_2) \subseteq Z'}$ and $Z' \preceq Y'$

$Z_1 \preceq \bot$ — Lemma 6.7.11

$(Z_1,Z_2) \preceq \bot$ — Rule for empty first component

$\bot \subseteq Y'$ — Rule for $\bot$

Case: $\dfrac{Z_2 \subseteq \bot}{(Z_1,Z_2) \subseteq Z'}$ and $Z' \preceq Y'$

Dual to the previous case.

Case: $\dfrac{}{\bot \subseteq Z'}$ and $Z' \preceq Y'$

Then $\bot \preceq \bot$ (rule for $\bot$) and $\bot \subseteq Y'$ (rule for $\bot$).

Case: $\dfrac{Z_1 \subseteq Z' \qquad Z_2 \subseteq Z'}{Z_1 \sqcup Z_2 \subseteq Z'}$ and $Z' \preceq Y'$

$Z_1 \preceq Y_1$ and $Y_1 \subseteq Y$ for some Y_1 — Ind. hyp.

$Z_2 \preceq Y_2$ and $Y_2 \subseteq Y$ for some Y_2 — Ind. hyp.

$Z_1 \sqcup Z_2 \preceq Y_1 \sqcup Y_2$ — Rule for $\sqcup$

$Y_1 \sqcup Y_2 \subseteq Y$ — Rule for $\sqcup$

Case: $\dfrac{Z \subseteq Z_1'}{Z \subseteq Z_1' \sqcup Z_2'}$ and $\dfrac{Z_1' \preceq Y_1' \qquad Z_2' \preceq Y_2'}{Z_1' \sqcup Z_2' \preceq \sqcup\{R \mid R \text{ in } Y_1' \text{ or } Y_2'\}}$

$Z \preceq Y$ and $Y \subseteq Y_1'$ for some Y — Ind. hyp.

$Y \subseteq Y_1' \sqcup Y_2'$ — Rule for $\sqcup$

Case: $\dfrac{Z \subseteq Z_2'}{Z \subseteq Z_1' \sqcup Z_2'}$ and $\dfrac{Z_1' \preceq Y_1' \qquad Z_2' \preceq Y_2'}{Z_1' \sqcup Z_2' \preceq \sqcup\{R \mid R \text{ in } Y_1' \text{ or } Y_2'\}}$

Dual to the previous case.

□

We now prove two small lemmas for $\subseteq$: that it is reflexive, and that empty pattern sorts (i.e., those with $Z \subseteq \bot$) behave like $\bot$ in sort assignment derivations.

Lemma 6.7.14 (Reflexivity of $\subseteq$)
For all Z, $Z \subseteq Z$.

Proof: By induction on Z.
The cases for R, cZ, $()$, and (Z_1,Z_2) and $\bot$ follow by the corresponding rules (i.e. the rules not involving $\bot$ in the premise). The case for $\sqcup$ follows.

Case: $Z = Z_1 \sqcup Z_2$

$Z_1 \subseteq Z_1$	and	$Z_2 \subseteq Z_2$	Ind. hyp.
$Z_1 \subseteq Z_1 \sqcup Z_2$	and	$Z_2 \subseteq Z_1 \sqcup Z_2$	Right rules for $\sqcup$
$Z_1 \sqcup Z_2 \subseteq Z_1 \sqcup Z_2$			Left rule for $\sqcup$

□

Lemma 6.7.15 (Correctness of Emptiness via $\subseteq$)
If $Z \subseteq \bot$ then $\Delta, P{\in}Z \vdash M \in R$ holds for any Δ, P, M and R.
(Provided $\Delta, P{\in}Z \vdash M \in R$ is well-formed).

Proof: By induction on the structure of the derivation of $Z \subseteq \bot$.
We first treat the cases where P is a variable or Z is a sort, and then treat the remaining cases following the derivation of $Z \subseteq \bot$.

Case: $P = x$

$\Delta, x{\in}\bot \vdash M \in R$	Rule for $\bot$
$\Delta, x{\in}Z \vdash M \in R$	Left subsumption for x

Case: $\dfrac{S \trianglelefteq \cdot}{S \subseteq \bot}$

$\Delta, P{\in}\bot \vdash M \in R$	Rule for $\bot$
$S \preceq \bot$	Rule for empty sort
$\Delta, P{\in}S \vdash M \in R$	Left subsumption for R

Case: $\dfrac{Z \subseteq \bot}{cZ \subseteq \bot}$ and $P = cP'$

$\Delta, P'{\in}Z \vdash M \in R$	Ind. hyp.
$\Delta, cP'{\in}cZ \vdash M \in R$	Rule for $cP{\in}cZ$

Case: $\dfrac{Z \subseteq \bot}{cZ \subseteq \bot}$ and $P = c'P'$ and $c' \neq c$

$\Delta, c'P'{\in}cZ \vdash M \in R$	Rule for $c'P{\in}cZ$

Case: $\dfrac{Z_1 \subseteq \bot}{(Z_1,Z_2) \subseteq \bot}$ and $P = (P_1,P_2)$

$\Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash M \in R$	Ind. hyp.
$\Delta, (P_1,P_2){\in}(Z_1,Z_2) \vdash M \in R$	Rule for pairs

Case: $\dfrac{Z_2 \subseteq \bot}{(Z_1,Z_2) \subseteq \bot}$ and $P = (P_1,P_2)$

Dual to the previous case.

Case: $\dfrac{}{\bot \subseteq \bot}$

$\Delta, P{\in}\bot \vdash M \in R$ Rule for $\bot$

Case: $\dfrac{Z_1 \subseteq \bot \quad Z_2 \subseteq \bot}{Z_1 \sqcup Z_2 \subseteq \bot}$

$\Delta, P{\in}Z_1 \vdash M \in R$ Ind. hyp.

$\Delta, P{\in}Z_2 \vdash M \in R$ Ind. hyp.

$\Delta, P{\in}Z_1 \sqcup Z_2 \vdash M \in R$ Rule for $\sqcup$

□

We now use the preceding lemmas to prove a subsumption lemma for the compositional inclusion judgment $Z \subseteq Z'$. The proof requires the induction hypothesis to be generalized to allow a context Δ to be replaced by any $\Delta' \subseteq \Delta$, which is defined pointwise, meaning that if $P{\in}Z$ in Δ then $P{\in}Z'$ in Δ' for some $Z' \subseteq Z$.

This lemma is required to show completeness when checking the body of a branch $P \Rightarrow M$: the assumptions Δ arising from P during checking will satisfy $\Delta \subseteq \Delta'$, where Δ' are the corresponding assumptions in any declarative sort assignment derivation.

This lemma is also interesting because it validates forms of reasoning that are even more flexible than those allowed by the rules of the declarative system: we can think of subsumption via $\subseteq$ as an admissible rule. We could have included the rules of $\subseteq$ directly in the original declarative system, but doing so would result in a relatively large system. Also, it would have added to the complexity of the earlier proofs, particularly for sort preservation and progress.

Lemma 6.7.16 (Subsumption for $\subseteq$)
If $\Delta \subseteq \Delta'$ then the following hold.

1. *If $\Delta' \vdash Z \bowtie \Omega \in R$ then $\Delta \vdash Z \bowtie \Omega \in R$.*
2. *If $\Delta' \vdash M \in R$ then $\Delta \vdash M \in R$.*

Proof: By induction on the structure of the sort assignment derivation, and the sum of the sizes of the compositional inclusion derivations in $\Delta \subseteq \Delta'$, lexicographically.

We first treat the cases where one of the derivations in $\Delta \subseteq \Delta'$ involves one of the rules for $Z \subseteq Z'$ which do not require Z' to be of a particular form. We then treat the remaining cases, following structure of the sort assignment derivation.

The cases for the right rules simply rebuild the derivation, since they either do not involve the context, or only involve assumptions of the form $x{\in}R$. We show the cases for x, λ and **case** only; the others are similar.

Case: $\Delta \subseteq \Delta'$, $\dfrac{S \trianglelefteq \cdot}{S \subseteq Z'}$ and $\Delta', P{\in}Z' \vdash M \in R$

$S \preceq \bot$ — Lemma 6.7.11
$\Delta, P{\in}\bot \vdash M \in R$ — Rule for $\bot$
$\Delta, P{\in}S \vdash M \in R$ — Left subsumption rule

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z \subseteq \bot}{cZ \subseteq Z'}$ and $\Delta', P{\in}Z' \vdash M \in R$

$Z \subseteq \bot$ — Rule for c with $\bot$
$\Delta, P{\in}Z \vdash M \in R$ — Lemma 6.7.15

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z_1 \subseteq \bot}{(Z_1,Z_2) \subseteq Z'}$ and $\Delta', P{\in}Z' \vdash M \in R$

$(Z_1,Z_2) \subseteq \bot$ — Rule for empty first comp.
$\Delta, P{\in}(Z_1,Z_2) \vdash M \in R$ — Lemma 6.7.15

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z_2 \subseteq \bot}{(Z_1,Z_2) \subseteq Z'}$ and $\Delta', P{\in}Z' \vdash M \in R$

Dual to the previous case.

Case: $\Delta \subseteq \Delta'$, $\dfrac{}{\bot \subseteq Z'}$ and $\Delta', P{\in}Z' \vdash M \in R$

$\Delta, P{\in}\bot \vdash M \in R$ — Rule for $\bot$.

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z_1 \subseteq Z' \quad Z_2 \subseteq Z'}{Z_1 \sqcup Z_2 \subseteq Z'}$ and $\Delta', P{\in}Z' \vdash M \in R$

$\Delta, P{\in}Z_1 \vdash M \in R$ — Ind. hyp.
$\Delta, P{\in}Z_2 \vdash M \in R$ — Ind. hyp.
$\Delta, P{\in}Z_1 \sqcup Z_2 \vdash M \in R$ — Rule for $\sqcup$

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z_1 \subseteq Z_1' \quad Z_2 \subseteq Z_2'}{(Z_1,Z_2) \subseteq (Z_1',Z_2')}$ and $\dfrac{\Delta', P_1{\in}Z_1', P_2{\in}Z_2' \vdash M{\in}R}{\Delta', (P_1, P_2) \in (Z_1',Z_2') \vdash M{\in}R}$

$\Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash M{\in}R$ — Ind. hyp.
$\Delta, (P_1, P_2){\in}(Z_1',Z_2') \vdash M{\in}R$ — Left rule for pair

Case: $\Delta \subseteq \Delta'$, $\dfrac{S_1{\times}S_2 \trianglelefteq S_1'{\times}S_2'}{S_1{\times}S_2 \subseteq S_1'{\times}S_2'}$ and $\dfrac{\Delta, P_1{\in}S_1', P_2{\in}S_2' \vdash M{\in}R}{\Delta, (P_1, P_2) \in S_1'{\times}S_2' \vdash M{\in}R}$

$(S_1 \trianglelefteq S_1'$ and $S_2 \trianglelefteq S_2')$
or $S_1 \trianglelefteq \cdot$ or $S_2 \trianglelefteq \cdot$ — Prod. Inv. Lemma 6.6.6

Subcase: $S_1 \trianglelefteq S_1'$ and $S_2 \trianglelefteq S_2'$

$\Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash M{\in}R$ — Ind. hyp.
$\Delta, (P_1, P_2){\in}S_1{\times}S_2 \vdash M{\in}R$ — Left rule for $\times$

Subcase: $S_1 \trianglelefteq \cdot$ (the subcase for $S_2 \trianglelefteq \cdot$ is dual)

$S_1 \subseteq \bot$ — Rule for empty sort
$\Delta, P_1 \in S_1, P_2 \in S_2 \vdash M \in R$ — Lemma 6.7.15
$\Delta, (P_1, P_2) \in S_1 \times S_2 \vdash M \in R$ — Left rule for $\times$

Case: $\Delta \subseteq \Delta'$, $\dfrac{}{() \subseteq ()}$ and $\dfrac{\Delta' \vdash M \in R}{\Delta', () \in () \vdash M \in R}$

$\Delta \vdash M \in R$ — Ind. hyp.
$\Delta, () \in () \vdash M \in R$ — Left rule for $()$

Case: $\Delta \subseteq \Delta'$, $\dfrac{S \trianglelefteq 1}{S \subseteq 1}$ and $\dfrac{\Delta' \vdash M \in R}{\Delta', () \in 1 \vdash M \in R}$

$\Delta \vdash M \in R$ — Ind. hyp.
$\Delta, () \in 1 \vdash M \in R$ — Left rule for 1
$\Delta, () \in S \vdash M \in R$ — Left subsumption rule

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z \subseteq Z'}{cZ \subseteq cZ'}$ and $\dfrac{\Delta', P \in Z' \vdash M \in R}{\Delta', cP \in cZ' \vdash M \in R}$

$\Delta, P \in Z \vdash M \in R$ — Ind. hyp.
$\Delta, cZ \in cP \vdash M \in R$ — Left rule for $cP \in cZ$

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z \subseteq Z'}{cZ \subseteq cZ'}$ and $\dfrac{c' \neq c}{\Delta', c'P \in cZ' \vdash M \in R}$

$\Delta, c'P \in cZ \vdash M \in R$ — Left rule for $c'P \in cZ$

Case: $\Delta \subseteq \Delta'$, $\dfrac{\rho \trianglelefteq \rho'}{\rho \subseteq \rho'}$ and $\dfrac{\Delta', cP \in \mathsf{inv}(\rho') \vdash M \in R}{\Delta', cP \in \rho' \vdash M \in R}$

For each $c'S$ in $\mathsf{inv}(\rho)$:

$c'S'$ in $\mathsf{inv}(\rho)$ with $S \trianglelefteq S'$ — Mono. of inv (6.6.10)
$c'S \subseteq c'S'$ — Rule for c, via rule for sorts
$c'S \subseteq \mathsf{inv}(\rho)$ — Right rule for $\sqcup$, repeatedly

$\mathsf{inv}(\rho) \subseteq \mathsf{inv}(\rho')$ — Left rule for $\sqcup$, repeatedly
$\Delta, cP \in \mathsf{inv}(\rho) \vdash M \in R$ — Ind. hyp.
$\Delta, cP \in \rho \vdash M \in R$ — Left rule for $cP \in \rho$

Case: $\Delta \subseteq \Delta'$, $Z \subseteq \bot$ and $\dfrac{}{\Delta', P \in \bot \vdash M \in R}$

All cases for $Z \subseteq \bot$ already considered.

Case: $\Delta \subseteq \Delta'$, $\dfrac{Z \subseteq Z_1'}{Z \subseteq Z_1' \sqcup Z_2'}$ and $\dfrac{\Delta', P \in Z_1' \vdash M \in R \qquad \Delta', P \in Z_2' \vdash M \in R}{\Delta', P \in Z_1' \sqcup Z_2' \vdash M \in R}$

$\Delta, P \in Z \vdash M \in R$ — Ind. hyp.

Case: $\Delta \subseteq \Delta', \dfrac{Z \subseteq Z_2'}{Z \subseteq Z_1' \sqcup Z_2'}$ and $\dfrac{\Delta', P{\in}Z_1' \vdash M{\in}R \qquad \Delta', P{\in}Z_2' \vdash M{\in}R}{\Delta', P \in Z_1' \sqcup Z_2' \vdash M{\in}R}$

$\Delta, P{\in}Z \vdash M{\in}R$ Ind. hyp.

Case: $\Delta \subseteq \Delta'$, $Z \subseteq Z'$ and $\dfrac{Z' \preceq Y' \qquad \Delta', x{\in}Y' \vdash M{\in}R}{\Delta', x{\in}Z' \vdash M{\in}R}$

$Z \preceq Y$ and $Y \subseteq Y'$ for some Y Lemma 6.7.13

$\Delta, x{\in}Y \vdash M{\in}R$ Ind. hyp.

$\Delta, x{\in}Z \vdash M{\in}R$ Left subsumption rule for x

Case: $\Delta \subseteq \Delta', \dfrac{S \trianglelefteq S'}{S \subseteq S'}$ and $\dfrac{S' \preceq Y' \qquad \Delta', P{\in}Y' \vdash M{\in}R}{\Delta', P{\in}S' \vdash M{\in}R}$

$\Delta, x{\in}Y' \vdash M{\in}R$ Ind. hyp.

$\Delta, x{\in}S' \vdash M{\in}R$ Left subsumption rule for sorts

$\Delta, x{\in}S \vdash M{\in}R$ Left subsumption rule for sorts

Case: $\Delta \subseteq \Delta', \dfrac{S \trianglelefteq S'}{S \subseteq S'}$ and $\dfrac{x{\in}S' \text{ in } \Delta', x{\in}S'}{\Delta', x{\in}S' \vdash x \in S'}$

$\Delta, x{\in}S \vdash x \in S$ Rule for x

$\Delta, x{\in}S \vdash x \in S'$ Right subsumption rule

Case: $\Delta \subseteq \Delta'$ and $\dfrac{\Delta', x{\in}S \vdash M \in R}{\Delta' \vdash \lambda x.M \in S \to R}$

$\Delta, x{\in}S \subseteq \Delta', x{\in}S$ Def. $\Delta \subseteq \Delta'$, Reflexivity

$\Delta, x{\in}S \vdash M \in R$ Ind. hyp.

$\Delta \vdash \lambda x.M \in S \to R$ Rule for λ

Case: $\Delta \subseteq \Delta'$ and $\dfrac{\Delta' \vdash M \in R \qquad \Delta' \vdash R \bowtie \Omega \in S}{\Delta' \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S}$

$\Delta \vdash M \in R$ Ind. hyp.

$\Delta \vdash R \bowtie \Omega \in S$ Ind. hyp.

$\Delta \vdash \mathbf{case}\ M\ \mathbf{of}\ \Omega \in S$ Rule for **case**

Case: $\Delta \subseteq \Delta'$ and $\dfrac{\Delta', P{\in}Z \vdash M{\in}R \quad \Delta' \vdash (Z \backslash P) \bowtie \Omega \in R}{\Delta' \vdash Z \bowtie (P \Rightarrow M \mid \Omega) \in R}$

$Z \subseteq Z$ Lemma 6.7.14

$\Delta, P{\in}Z \subseteq \Delta', P{\in}Z$ Def. $\Delta \subseteq \Delta'$

$\Delta, P{\in}Z \vdash M{\in}R$ Ind. hyp.

$\Delta \vdash (Z \backslash P) \bowtie \Omega \in R$ Ind. hyp.

$\Delta \vdash Z \bowtie (P \Rightarrow M \mid \Omega) \in R$ Rule for non-empty branches

Case: $\Delta \subseteq \Delta'$ and $\dfrac{Z \preceq \bot}{\Delta' \vdash Z \bowtie \cdot \in R}$

$\Delta \vdash Z \bowtie \cdot \in R$ Rule for empty Z

□

We now combine the previous lemma with the Preservation of $\subseteq$ Lemma (6.7.10) to prove a lemma validating a form of subsumption that involves the sort of the case object in the sort assignment judgment for branches. This is a key lemma in the completeness proof.

Lemma 6.7.17 (Subsumption for case objects)
If $Z \subseteq Z'$ and $\Delta \vdash Z' \bowtie \Omega \in R$ then $\Delta \vdash Z \bowtie \Omega \in R$.

Proof: By induction on the structure of the derivation of $\Delta \vdash Z' \bowtie \Omega \in R$.

Case: $\dfrac{\Delta, P{\in}Z' \vdash M{\in}R \qquad \Delta \vdash (Z'\backslash P) \bowtie \Omega \in R}{\Delta \vdash Z' \bowtie (P \Rightarrow M \mid \Omega) \in R}$

$\Delta, P{\in}Z \subseteq \Delta, P{\in}Z'$ Def. $\Delta \subseteq \Delta'$

$\Delta, P{\in}Z \vdash M{\in}R$ Ind. hyp.

$Z\backslash P \subseteq Z'\backslash P$ Preservation of $\subseteq$ (6.7.10)

$\Delta \vdash (Z\backslash P) \bowtie \Omega \in R$ Ind. hyp.

$\Delta \vdash Z \bowtie (P \Rightarrow M \mid \Omega) \in R$ Rule for non-empty branches

Case: $\dfrac{Z' \preceq \bot}{\Delta \vdash Z' \bowtie \cdot \in R}$

$Z \preceq Y$ and $Y \subseteq \bot$ for some $Y = R_1 \sqcup \ldots \sqcup R_n$
with $R_i \not\trianglelefteq \cdot$ for each R_i in Y Lemma 6.7.13

$Y \preceq \bot$ Lemma 6.7.11

$R_i \trianglelefteq \cdot$ for all R_i in Y Inversion on $Y \preceq \bot$, repeatedly

$Y = \bot$ $R_i \trianglelefteq \cdot$ contradicts $R_i \not\trianglelefteq \cdot$

$\Delta \vdash Z \bowtie \cdot \in R$ Rule for empty Z

□

The next two lemmas capture the completeness of the auxiliary judgments $A \stackrel{\Rightarrow}{\bot} S$ and $R \stackrel{\Rightarrow}{\trianglelefteq} S$.

Lemma 6.7.18 (Completeness of $\stackrel{\Rightarrow}{\bot}$)
If $R \sqsubset A$ and $R \trianglelefteq \cdot$ then $A \stackrel{\Rightarrow}{\bot} S$ for some S.

Proof: By induction on the structure of A.

Case: $A = a$, $R = \rho$

For each s in ρ:

$s \sqsubset a$ Well formedness of ρ

$\&\{r \mid r \sqsubset a\} \trianglelefteq s$ $\&$ is g.l.b. (5.8.11)

$\&\{r \mid r \sqsubset a\} \trianglelefteq \rho$ $\&$ is g.l.b. (5.8.11)

$\rho \trianglelefteq \cdot$ Assumption

$\&\{r \mid r \sqsubset a\} \trianglelefteq \cdot$ Transitivity

$a \overset{\Rightarrow}{\perp} \&\{r \mid r \sqsubset a\}$ Rule for a

Case: $A = A_1 \times A_2$, $R \sqsubset A_1 \times A_2$

$R \overset{\Rightarrow}{\cong} R_1 \otimes R_2$ Lemma 5.6.8

$R_1 \times R_2 \trianglelefteq \cdot$ Transitivity

$R_1 \trianglelefteq \cdot$ or $R_2 \trianglelefteq \cdot$ Inversion on $R_1 \times R_2 \trianglelefteq \cdot$

Subcase: $R_1 \trianglelefteq \cdot$ (the subcase for $R_2 \trianglelefteq \cdot$ is dual)

$A_1 \overset{\Rightarrow}{\perp} S_1$ for some $S_1 \trianglelefteq \cdot$ Ind. hyp.

$A_1 \times A_2 \overset{\Rightarrow}{\perp} S_1 \times \top^{A_2}$ Rule for first component empty

Case: $A = A_1 \to A_2$

Impossible: no rule matches $R \trianglelefteq \cdot$ with $R \sqsubset A_1 \to A_2$.

□

Lemma 6.7.19 (Completeness of $\overset{\Rightarrow}{\trianglelefteq}$)
If $R \trianglelefteq S_1 \to S_2$ then $R \overset{\Rightarrow}{\trianglelefteq} R_1 \to R_2$ with $S_1 \trianglelefteq R_1$ and $R_2 \trianglelefteq S_2$.

Proof: By induction on the structure of the derivation of $R \trianglelefteq S_1 \to S_2$. The only rules matching a conclusion of the form $R \trianglelefteq S_1 \to S_2$ are the following.

Case: $\dfrac{S_1 \trianglelefteq R_1 \qquad R_2 \trianglelefteq S_2}{R_1 \to R_2 \trianglelefteq S_1 \to S_2}$ Then $R_1 \to R_2 \overset{\Rightarrow}{\trianglelefteq} R_1 \to R_2$ (rule for $\to$).

Case: $\dfrac{R_1 \trianglelefteq S_1 \to S_2}{R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}$ Then $R_1 \overset{\Rightarrow}{\trianglelefteq} R_1 \to R_2$ (ind. hyp.)
so $R_1 \& R_2 \overset{\Rightarrow}{\trianglelefteq} R_1 \to R_2$ (rule for $\&$).

Case: $\dfrac{R_2 \trianglelefteq S_1 \to S_2}{R_1 \,\&\, R_2 \trianglelefteq S_1 \to S_2}$ Dual to the previous case.

□

We now use the preceding lemmas to prove the main theorem of this section: the completeness of the sort checking algorithm. Our choice to use inversion lemmas does not allow us to structure the proof of completeness by induction on the given derivation: we would not be able to apply the induction hypothesis to the result of the inversion lemma. In the completeness proofs in Chapter 2 and Chapter 3 we used induction on the structure of terms for this reason, but this is not sufficient here: the left rules deconstruct patterns in the context rather than terms. We thus require a more complicated induction ordering, and it suffices to use the same well order as used in the termination proof for the algorithm. Choosing this order also allows the structure of the completeness proof to follow the steps taken by the algorithm.

Theorem 6.7.20 (Completeness of Sort Checking)

1. *If* $\Pi \vdash I \in R$ *then* $\Pi \vdash I \stackrel{\Rightarrow}{\in} R'$ *for some* $R' \trianglelefteq R$.
2. *If* $\Pi \vdash C \in R$ *then* $\Pi \vdash C \stackrel{\Leftarrow}{\in} R$.
3. *If* $(\Pi, \Delta) \vdash C \in R$ *then* $\Pi\,;\, \Delta \vdash C \stackrel{\Leftarrow}{\in} R$.
4. *If* $\Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \in R$ *then* $\Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \stackrel{\Leftarrow}{\in} R$.

Proof:
By induction on the form of the conclusion of the algorithmic sort checking derivation, according to the same well ordering as the termination theorem. Thus, lexicographically on the following.

- The term I or C or the cases Ω^{c}.
- Which judgment the conclusion is an instance of (which allows each part to use earlier parts of the induction hypothesis).
- The sort R or S.
- The sum of the sizes of the patterns P in Δ.
- The number of occurrences of base sorts ρ in Δ not enclosed by a constructor c.
- The sum of the sizes of the pattern sorts Z in Δ.

For parts 1,2 and 4 we have one case for each construct matching the grammars for I, C and Ω^{c}, respectively. In part 3 we have one case for each possible form for the last assumption in Δ, and one case for when Δ is empty (hence all assumptions are in Π, and are of the form $x{\in}R$).

Almost all the cases have the same structure: they decompose the declarative sort assignment derivation using the appropriate inversion lemma, then apply the induction hypothesis to each part, as appropriate, and finally rebuild the derivation using the appropriate algorithmic rule.

Part 1: $\boxed{\text{If } \Pi \vdash I \in R \text{ then } \Pi \vdash I \stackrel{\Rightarrow}{\in} R' \text{ for some } R' \trianglelefteq R.}$

Case: $\Pi \vdash x \in R$

$x{\in}R'$ in Π for some $R' \trianglelefteq R$	Inv. Lemma (6.7.9, 1a)
$\Pi \vdash x \stackrel{\Rightarrow}{\in} R'$	Rule for x

Case: $\Pi \vdash u \in R$

$u{\in}R'$ in Π for some $R' \trianglelefteq R$	Inv. Lemma (6.7.9, 1b)
$\Pi \vdash u \stackrel{\Rightarrow}{\in} R'$	Rule for u

Case: $\Pi \vdash c\,C \in \rho$

cR' in $\mathsf{body}(\rho')$ and $\Pi \vdash C \in R'$ and $\rho' \trianglelefteq \rho$ for some R' and ρ'	Inv. Lemma (6.7.9, 1c)
$\Pi \vdash C \overset{\Leftarrow}{\in} R'$	Ind. hyp.
$\Pi \vdash cC \overset{\Rightarrow}{\in} \rho'$	Rule for c

Case: $\Pi \vdash I\,C \in R$

$\Pi \vdash I \in R_2 \to R$ and $\Pi \vdash C \in R_2$ for some R_2	Inv. Lemma (6.7.9, 1d)
$\Pi \vdash I \overset{\Rightarrow}{\in} S_1$ for some $S_1 \trianglelefteq R_2 \to R$	Ind. hyp.
$S_1 \overset{\Rightarrow}{\trianglelefteq} R_2' \to R'$ with $R_2 \trianglelefteq R_2'$ and $R' \trianglelefteq R$	Compl. of $\overset{\Rightarrow}{\trianglelefteq}$ (6.7.19)
$\Pi \vdash C \in R_2'$	Right subsumption rule
$\Pi \vdash C \overset{\Leftarrow}{\in} R_2'$	Ind. hyp.
$\Pi \vdash I\,C \overset{\Rightarrow}{\in} R'$	Rule for appl.

Case: $\Pi \vdash (C{\in}L) \in R$

$\Pi \vdash C \in R'$ for some R' in L with $R' \trianglelefteq R$	Inv. Lemma (6.7.9, 1e)
$\Pi \vdash C \overset{\Leftarrow}{\in} R'$	Ind. hyp
$\Pi \vdash (C{\in}L) \overset{\Rightarrow}{\in} R'$	Rule for annotation

Case: $\Pi \vdash () \in R$

$\mathsf{1} \trianglelefteq R$	Inv. Lemma (6.7.9, 1f)
$\Pi \vdash () \overset{\Rightarrow}{\in} \mathsf{1}$	Rule for $()$

Case: $\Pi \vdash \mathbf{fix}\ u.(C{\in}L) \in R$

$\Pi, u{\in}R' \vdash C \in R'$ for some R' in L with $R' \trianglelefteq R$	Inv. Lemma (6.7.9, 1g)
$\Pi, u{\in}R' \vdash C \overset{\Leftarrow}{\in} R'$	Ind. hyp.
$\Pi \vdash \mathbf{fix}\ u.(C{\in}L) \overset{\Rightarrow}{\in} R'$	Rule for **fix** with annot.

. . . . . .

Part 2: If $\Pi \vdash C \in R$ then $\Pi \vdash C \overset{\Leftarrow}{\in} R'$.

Case: $\Pi \vdash I \in R$

$\Pi \vdash I \overset{\Rightarrow}{\in} R'$ for some $R' \trianglelefteq R$	Ind. hyp (same I, earlier part)
$\Pi \vdash I \overset{\Leftarrow}{\in} R$	Rule for I

Case: $\Pi \vdash \lambda x.C \in \top$

$\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} \top$	Rule for λ with $\top$

Case: $\Pi \vdash \lambda x.C \in R_1 \,\&\, R_2$

$R_1 \,\&\, R_2 \trianglelefteq R_1$	$\&$ is g.l.b. (Lemma 5.8.11)
$\Pi \vdash \lambda x.C \in R_1$	Subsumption

$\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R_1$ — Ind. hyp. (same C, smaller R)

$\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R_2$ — Similarly, dual to the above

$\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R_1 \,\&\, R_2$ — Rule for λ with &

Case: $\Pi \vdash \lambda x.C \in R_1 \to R_2$

$R_1 \to R_2 \trianglelefteq R_1 \to R_2$ — Reflexivity

$\Pi, x{\in}R_1 \vdash C \in R_2$ — Inv. Lemma (6.7.9, 2a)

$\Pi, x{\in}R_1 \vdash C \overset{\Leftarrow}{\in} R_2$ — Ind. hyp.

$\Pi \vdash \lambda x.C \overset{\Leftarrow}{\in} R_1 \to R_2$ — Rule for λ with $\to$

Case: $\Delta \vdash (C_1, C_2) \in R$

$R \overset{\Rightarrow}{\cong} R_1 \otimes R_2$ for some R_1 and R_2 — Lemma 5.6.8

$\Pi \vdash C_1 \in S_1$ and $\Pi \vdash C_2 \in S_2$
for some S_1 and S_2 with $S_1 \times S_2 \trianglelefteq R_1 \times R_2$ — Inv. Lemma (6.7.9, 2b)

($S_1 \trianglelefteq R_1$ and $S_2 \trianglelefteq R_2$)
or $S_1 \trianglelefteq \cdot$ or $S_2 \trianglelefteq \cdot$ — Prod. Inv. Lemma 6.6.6

Subcase: $S_1 \trianglelefteq R_1$ and $S_2 \trianglelefteq R_2$

$\Pi \vdash C_1 \in R_1$ and $\Pi \vdash C_2 \in R_2$ — Subsumption rule

$\Pi \vdash C_1 \overset{\Leftarrow}{\in} R_1$ and $\Pi \vdash C_2 \overset{\Leftarrow}{\in} R_2$ — Ind. hyp.

$\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R$ — Rule for pairs

Subcase: $S_1 \trianglelefteq \cdot$ (the subcase for $S_2 \trianglelefteq \cdot$ is dual)

$A_1 \overset{\Rightarrow}{\perp} S_1'$ for some S_1' — Completeness of $\overset{\Rightarrow}{\perp}$ (6.7.18)

$[\![S_1]\!] \subseteq \{\}$ — Lemma 5.8.10

$[\![S_1]\!] \subseteq [\![S_1']\!]$ — $\{\}$ is minimal

$S_1 \trianglelefteq S_1'$ — Lemma 5.8.10

$\Pi \vdash C_1 \in S_1'$ — Subsumption rule

$\Pi \vdash C_1 \overset{\Leftarrow}{\in} S_1'$ — Ind. hyp. (smaller C)

$S_2 \trianglelefteq \top$ — Rule for $\top$

$\Pi \vdash C_2 \in \top$ — Subsumption rule

$\Pi \vdash C_2 \overset{\Leftarrow}{\in} \top$ — Ind. hyp. (smaller C)

$\Pi \vdash (C_1, C_2) \overset{\Leftarrow}{\in} R$ — Rule for first component empty

Case: $\Pi \vdash \mathbf{case}\ I\ \mathbf{of}\ \Omega^{\mathsf{c}} \in R$

$\Pi \vdash I \in S'$ and $\Pi \vdash S \bowtie \Omega^{\mathsf{c}} \in R$ for some S — Inv. Lemma (6.7.9, 2c)

$\Pi \vdash I \overset{\Rightarrow}{\in} S'$ with $S' \trianglelefteq S$ — Ind. hyp.

$\Pi \vdash S' \bowtie \Omega^{\mathsf{c}} \in R$ — Subs. for case objects (6.7.17)

$\Pi \vdash S' \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$ — Ind. Hyp.

$\Pi \vdash \mathbf{case}\ I\ \mathbf{of}\ \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$

Part 3: If $(\Pi, \Delta) \vdash C \in R$ then $\Pi\,;\, \Delta \vdash C \overset{\Leftarrow}{\in} R$.

Case: $\Pi, \Delta, P{\in}Z_1 \sqcup Z_2 \vdash C \in R$

$\Pi, \Delta, P{\in}Z_1 \vdash C \in R$ and $\Pi, \Delta, P{\in}Z_2 \vdash C \in R$	Inv. Lemma (6.7.9, 3a)
$\Pi; (\Delta, P{\in}Z_1) \vdash C \overset{\Leftarrow}{\in} R$ and $\Pi; (\Delta, P{\in}Z_2) \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (smaller Z)
$\Pi; (\Delta, P{\in}Z_1 \sqcup Z_2) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $\sqcup$

Case: $\Pi, \Delta, P{\in}\bot \vdash C \in R$

$\Pi; (\Delta, P{\in}\bot) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $\bot$

Case: $\Pi, \Delta, (P_1{,}P_2){\in}S \vdash C \in R$

$S \overset{\Rightarrow}{\cong} S_1 \otimes S_2$ and $\Pi, \Delta, P_1{\in}S_1, P_2{\in}S_2 \vdash C \in R$	Inv. Lemma (6.7.9, 3b)
$\Pi; (\Delta, P_1{\in}S_1, P_2{\in}S_2) \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (smaller P)
$\Pi; (\Delta, (P_1{,}P_2){\in}S) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $\times$

Case: $\Pi, \Delta, (P_1{,}P_2){\in}(Z_1{,}Z_2) \vdash C \in R$

$\Pi, \Delta, P_1{\in}Z_1, P_2{\in}Z_2 \vdash C \in R$	Inv. Lemma (6.7.9, 3c)
$\Pi; (\Delta, P_1{\in}Z_1, P_2{\in}Z_2) \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (smaller P)
$\Pi; (\Delta, (P_1{,}P_2){\in}(Z_1{,}Z_2)) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for pair

Case: $\Pi, \Delta, ()\in() \vdash C \in R$

$\Pi, \Delta \vdash C \in R$	Inv. Lemma (6.7.9, 3d)
$\Pi; \Delta \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (smaller total P in Δ)
$\Pi; (\Delta, ()\in()) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $()$

Case: $\Pi, \Delta, ()\in S \vdash C \in R$

$\Pi, \Delta \vdash C \in R$	Inv. Lemma (6.7.9, 3e)
$\Pi; \Delta \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (smaller total P in Δ)
$\Pi; (\Delta, ()\in S) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $()$ with S

Case: $\Pi, \Delta, cP{\in}cZ \vdash C \in R$

$\Pi, \Delta, P{\in}Z \vdash C \in R$	Inv. Lemma (6.7.9, 3f)
$\Pi; (\Delta, P{\in}Z) \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (smaller P)
$\Pi; (\Delta, cP{\in}cZ) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $cP{\in}cZ$

Case: $\Pi, \Delta, cP{\in}c'Z \vdash C \in R$

$\Pi; (\Delta, cP{\in}c'Z) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for $cP{\in}c'Z$

Case: $\Pi, \Delta, cP{\in}\rho \vdash C \in R$

$\Pi, \Delta, cP{\in}\mathsf{inv}(\rho) \vdash C \in R$	Inv. Lemma (6.7.9, 3g)
$\Pi; (\Delta, cP{\in}\mathsf{inv}(\rho)) \vdash C \overset{\Leftarrow}{\in} R$	Ind. Hyp. (fewer ρ not enclosed by c)
$\Pi; (\Delta, cP{\in}\mathsf{inv}(\rho)) \vdash C \overset{\Leftarrow}{\in} R$	Left rule for inv

Case: $\Pi, \Delta, x{\in}Z \vdash C \in R$

$Z \overset{\Rightarrow}{\preceq} Y$ and for each S in Y
we have $\Pi, \Delta, x{\in}S \vdash C \in R$ — Inv. Lemma (6.7.9, 3h)

For each S in Y we have:
$(\Pi, x{\in}S); \Delta \vdash C \overset{\Leftarrow}{\in} R$ — Ind. hyp. (smaller total P in Δ)

$\Pi; (\Delta, x{\in}Z) \vdash C \overset{\Leftarrow}{\in} R$ — Left rule for $x{\in}Z$

Case: $\Pi \vdash C \in R \quad (\text{with } \Delta = \cdot)$

$\Pi \vdash C \overset{\Leftarrow}{\in} R$ — Ind. hyp. (earlier part)

$\Pi; \cdot \vdash C \overset{\Leftarrow}{\in} R$ — Left rule for $\cdot$

. . . . . .

Part 4: $\boxed{\text{If } \Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \in R \text{ then } \Pi \vdash Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R.}$

Case: $\dfrac{\Pi, P{\in}Z \vdash C{\in}R \qquad \Delta \vdash (Z \backslash P) \bowtie \Omega^{\mathsf{c}} \in R}{\Delta \vdash Z \bowtie (P \Rightarrow C \mid \Omega^{\mathsf{c}}) \in R}$

$\Pi, P{\in}Z \vdash C \overset{\Leftarrow}{\in} R$ — Ind. hyp.

$\Pi \vdash (Z \backslash P) \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$ — Ind. hyp.

$\Pi \vdash Z \bowtie (P \Rightarrow C \mid \Omega^{\mathsf{c}}) \overset{\Leftarrow}{\in} R$ — Rule for non-empty branches

Case: $\dfrac{Z \preceq \bot}{\Pi \vdash Z \bowtie \cdot \in R}$

$Z \overset{\Rightarrow}{\preceq} Y$ with each S in Y satisfying
$S \trianglelefteq S'$ for some S' in $\bot$ — Compl. of $\overset{\Rightarrow}{\preceq}$ (6.7.8)

Y contains no S — $\bot$ contains no S'

$Y = \bot$

$\Pi \vdash Z \bowtie \cdot \overset{\Leftarrow}{\in} R$ — Rule for empty Z

□

6.7.6 Annotatability

We now extend the annotatability theorem from Chapter 3 to the additional constructs of this chapter. The appropriate definitions of minimal inferable and checkable terms are as follows.

Definition 6.7.21

$$\begin{array}{rrl}
\textit{Minimal Inferable Terms} & \overset{\mathsf{m}}{I} & ::= x \mid u \mid c\,\overset{\mathsf{m}}{C} \mid \overset{\mathsf{m}}{I}\,\overset{\mathsf{m}}{C} \mid (\lambda x{:}A.\overset{\mathsf{m}}{C}){\in}L \\
& & \mid \mathbf{fix}\,u{:}A.(\overset{\mathsf{m}}{C}{\in}L) \mid () \mid (\overset{\mathsf{m}}{C},\overset{\mathsf{m}}{C}){\in}L \\
& & \mid (\mathbf{case}\,\overset{\mathsf{m}}{I}\,\mathbf{of}\,\overset{\mathsf{m}}{\Omega}){\in}L \\
\textit{Minimal Checkable Terms} & \overset{\mathsf{m}}{C} & ::= x \mid u \mid c\,\overset{\mathsf{m}}{C} \mid \overset{\mathsf{m}}{I}\,\overset{\mathsf{m}}{C} \mid \lambda x{:}A.\overset{\mathsf{m}}{C} \\
& & \mid \mathbf{fix}\,u{:}A.(\overset{\mathsf{m}}{C}{\in}L) \mid () \mid (\overset{\mathsf{m}}{C},\overset{\mathsf{m}}{C}) \\
& & \mid \mathbf{case}\,\overset{\mathsf{m}}{I}\,\mathbf{of}\,\overset{\mathsf{m}}{\Omega} \\
\textit{Minimal Branches} & \overset{\mathsf{m}}{\Omega} & ::= \cdot \mid (P{\Rightarrow}\overset{\mathsf{m}}{C} \mid \overset{\mathsf{m}}{\Omega})
\end{array}$$

The annotation combination functions extend as expected. We also require an annotation combination function for minimal branches, which we also write as $\overset{\mathsf{m}}{\Omega}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{\Omega}_2$. Its definition is purely compositional, and so it is not explicitly given here. Similarly, the compositional cases for $\overset{\mathsf{i}}{\bowtie}$ and $\overset{\mathsf{c}}{\bowtie}$ are omitted here. The interesting cases are the non-compositional cases for $\overset{\mathsf{i}}{\bowtie}$ and $\overset{\mathsf{c}}{\bowtie}$, which are the ones involving annotations. These cases are as follows.

$$\begin{array}{rcl}
((\lambda x.\overset{\mathsf{m}}{C}_1){\in}L_1) \overset{\mathsf{i}}{\bowtie} ((\lambda x.\overset{\mathsf{m}}{C}_2){\in}L_2) & = & (\lambda x.\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2){\in}L_1,L_2 \\
\mathbf{fix}\,u.(\overset{\mathsf{m}}{C}_1{\in}L_1) \overset{\mathsf{i}}{\bowtie} \mathbf{fix}\,u.(\overset{\mathsf{m}}{C}_2{\in}L_2) & = & \mathbf{fix}\,u.(\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2 \in L_1,L_2) \\
((\overset{\mathsf{m}}{C}_{11},\overset{\mathsf{m}}{C}_{12}){\in}L_1) \overset{\mathsf{i}}{\bowtie} ((\overset{\mathsf{m}}{C}_{21},\overset{\mathsf{m}}{C}_{22}){\in}L_2) & = & (\overset{\mathsf{m}}{C}_{11} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_{21},\overset{\mathsf{m}}{C}_{12} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_{22}) \in L_1,L_2 \\
((\mathbf{case}\,\overset{\mathsf{m}}{I}_1\,\mathbf{of}\,\overset{\mathsf{m}}{\Omega}_1){\in}L_1) \overset{\mathsf{i}}{\bowtie} ((\mathbf{case}\,\overset{\mathsf{m}}{I}_2\,\mathbf{of}\,\overset{\mathsf{m}}{\Omega}_2){\in}L_2) & = & (\mathbf{case}\,\overset{\mathsf{m}}{I}_1 \overset{\mathsf{i}}{\bowtie} \overset{\mathsf{m}}{I}_2\,\mathbf{of}\,\overset{\mathsf{m}}{\Omega}_1 \overset{\mathsf{i}}{\bowtie} \overset{\mathsf{m}}{\Omega}_2) \in L_1,L_2 \\
\mathbf{fix}\,u.(\overset{\mathsf{m}}{C}_1{\in}L_1) \overset{\mathsf{c}}{\bowtie} \mathbf{fix}\,u.(\overset{\mathsf{m}}{C}_2{\in}L_2) & = & \mathbf{fix}\,u.(\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2 \in L_1,L_2)
\end{array}$$

The annotation combination lemma extends without difficulty, using an additional part for minimal branches, as follows.

Lemma 6.7.22 (Annotation Combination)

1. *If* $\|\overset{\mathsf{m}}{I}_1\| = \|\overset{\mathsf{m}}{I}_2\|$ *and either* $\Delta \vdash \overset{\mathsf{m}}{I}_1 \in R$ *or* $\Delta \vdash \overset{\mathsf{m}}{I}_2 \in R$ *then* $\Delta \vdash (\overset{\mathsf{m}}{I}_1 \overset{\mathsf{i}}{\bowtie} \overset{\mathsf{m}}{I}_2) \in R$.
2. *If* $\|\overset{\mathsf{m}}{C}_1\| = \|\overset{\mathsf{m}}{C}_2\|$ *and either* $\Delta \vdash \overset{\mathsf{m}}{C}_1 \in R$ *or* $\Delta \vdash \overset{\mathsf{m}}{C}_2 \in R$ *then* $\Delta \vdash (\overset{\mathsf{m}}{C}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C}_2) \in R$.
3. *If* $\|\overset{\mathsf{m}}{\Omega}_1\| = \|\overset{\mathsf{m}}{\Omega}_2\|$ *and either* $\Delta \vdash Z \bowtie \overset{\mathsf{m}}{\Omega}_1 \in R$ *or* $\Delta \vdash Z \bowtie \overset{\mathsf{m}}{\Omega}_2 \in R$ *then* $\Delta \vdash Z \bowtie (\overset{\mathsf{m}}{\Omega}_1 \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{\Omega}_2) \in R$.

Proof: By induction on the structure of the sort assignment derivation. All of the additional cases are completely straightforward, except for the cases for **fix**. The following is the case for **fix** with $\overset{\mathsf{i}}{\bowtie}$ and $\Delta \vdash \overset{\mathsf{m}}{I}_1 \in R$; the case for **fix** with $\overset{\mathsf{i}}{\bowtie}$ $\Delta \vdash \overset{\mathsf{m}}{I}_2 \in R$ is dual, and the cases for **fix** with $\overset{\mathsf{c}}{\bowtie}$ are identical except that $\overset{\mathsf{i}}{\bowtie}$ is replaced by $\overset{\mathsf{c}}{\bowtie}$ in the last step.

Case: $\dfrac{\Delta, u{\in}R \vdash (\overset{\mathsf{m}}{C_1}{\in}L_1) \in R}{\Delta \vdash \mathbf{fix}\, u.(\overset{\mathsf{m}}{C_1}{\in}L_1) \in R}$ with $\mathbf{fix}\, u.(\overset{\mathsf{m}}{C_2}{\in}L_2)$ and $\|\overset{\mathsf{m}}{C_1}\| = \|\overset{\mathsf{m}}{C_2}\|$

$\|\overset{\mathsf{m}}{C_1}{\in}L_1\| = \|\overset{\mathsf{m}}{C_2}{\in}L_2\|$	Def. $\|\cdot\|$
$\Delta, u{\in}R \vdash (\overset{\mathsf{m}}{C_1}{\in}L_1) \overset{\mathsf{i}}{\bowtie} (\overset{\mathsf{m}}{C_2}{\in}L_2) \in R$	By ind. hyp.
$(\overset{\mathsf{m}}{C_1}{\in}L_1) \overset{\mathsf{i}}{\bowtie} (\overset{\mathsf{m}}{C_2}{\in}L_2) = ((\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2}){\in}L_1, L_2)$	By def. $\overset{\mathsf{i}}{\bowtie}, \overset{\mathsf{c}}{\bowtie}$
$\Delta \vdash \mathbf{fix}\, u.(\overset{\mathsf{m}}{C_1} \overset{\mathsf{c}}{\bowtie} \overset{\mathsf{m}}{C_2} \in L_1{,}L_2) \in R$	Rule for **fix**
$\Delta \vdash (\mathbf{fix}\, u.(\overset{\mathsf{m}}{C_1}{\in}L_1)) \overset{\mathsf{i}}{\bowtie} (\mathbf{fix}\, u.(\overset{\mathsf{m}}{C_2}{\in}L_2)) \in R$	Def. of $\overset{\mathsf{i}}{\bowtie}$

□

Finally, we have the main annotatability theorem, which also extends without difficulty once we add an additional part for minimal branches.

Theorem 6.7.23 (Annotatability)

1. *If $\Delta \vdash M \in R$ then we can construct a minimal inferable term $\overset{\mathsf{m}}{I}$ and a minimal checkable term $\overset{\mathsf{m}}{C}$ such that $\|\overset{\mathsf{m}}{I}\| = M$ and $\Delta \vdash \overset{\mathsf{m}}{I} \in R$ and $\|\overset{\mathsf{m}}{C}\| = M$ and $\Delta \vdash \overset{\mathsf{m}}{C} \in R$.*

2. *If $\Delta \vdash Z \bowtie \Omega \in R$ then we can construct a minimal branches expression $\overset{\mathsf{m}}{\Omega}$ such that $\|\overset{\mathsf{m}}{\Omega}\| = \Omega$ and $\Delta \vdash Z \bowtie \overset{\mathsf{m}}{\Omega} \in R$.*

Proof: By a straightforward induction on the structure of the sort assignment derivation, using the previous lemma to combine annotations. The following is the case for **fix**.

Case: $\dfrac{\Delta, u{\in}R \vdash M \in R}{\Delta \vdash \mathbf{fix}\, u.M \in R}$

$\Delta, u{\in}R \vdash \overset{\mathsf{m}}{C} \in R$ for some $\overset{\mathsf{m}}{C}$ with $\|\overset{\mathsf{m}}{C}\| = M$	Ind. hyp.
$\Delta, u{\in}R \vdash (\overset{\mathsf{m}}{C}{\in}R) \in R$	Rule for annotation
$\Delta \vdash \mathbf{fix}\, u.(\overset{\mathsf{m}}{C}{\in}R) \in R$	Rule for **fix**
$\|\mathbf{fix}\, u.(\overset{\mathsf{m}}{C}{\in}R)\| = \mathbf{fix}\, u.M$	Def. $\|\cdot\|$
$\mathbf{fix}\, u.(\overset{\mathsf{m}}{C}{\in}R)$ is minimal inferable and minimal checkable	Def. $\overset{\mathsf{m}}{I}$ and $\overset{\mathsf{m}}{C}$

□

Chapter 7

Extending to Standard ML

Standard ML includes many, many features beyond the language treated in the previous chapters. In this chapter we consider a number of issues that arise when extending sort checking to the full SML language. An additional aim of this chapter is to provide a sufficient level of detail to allow the reader to write programs in the language accepted by the sort checker, while leaving the details of the implementation of the sort checker until Chapter 8.

7.1 Annotation syntax

The annotations required for sort checking are contained within special comments such as the those in the following code.

```
(*[  datasort tt = true  ]*)

(*[  sortdef twott = tt * tt  ]*)

( (fn x => x) (*[ <: tt -> tt ]*)  ) true

(*[  val f <: tt -> tt  ]*)
fun f true = true
```

Any comment that begins with `(*[` and ends with `]*)` is treated as an annotation rather than a comment by the sort checker. This choice means that the annotations will be ignored by SML implementations that are unaware of sorts, and so programs can be compiled and executed without removing the annotations.

The first annotation above is a datasort declaration that declares a refinement of the SML datatype `bool`. The syntax of datasort declarations closely mirrors the syntax of Standard ML datatype declarations.

The second annotation is a sort abbreviation. It defines `twott` to be an abbreviation for the sort `tt * tt`. This form of declaration is very similar to SML `type` declarations. We use the keyword `sortdef` rather than `sort` because the latter seems likely to be a commonly used variable name in existing SML programs.

The third annotation is a sort annotation: the expression `(fn x => x)` is annotated with the sort `tt -> tt`. Sort annotations may appear after expressions, or within patterns. Sort

annotations can include comma-separated alternatives, as in the preceding chapters, in which case each alternative is tried. Sort annotations at the top level of patterns in value declarations are treated as though they were attached to the expression, in order to make the expression an inferable form. Other sort annotations are treated as bounds: every case matching the pattern must have a subsort of the bound.

The fourth annotation is a value sort specification. Value sort specifications are similar to sort annotations, but apply to any following SML `fun` or `val` declarations which bind the variable named in the annotation (`f` in this case). We allow the keyword `val` to be omitted in the case when an annotation comment contains only value sort specifications. This is convenient because it allows the indentation of these specifications to match that of the declaration which follows, as in the example below.

Value sort specifications allow the sorts of mutually recursive functions to be declared by having a sequence of value sort specifications prior to the function definitions. This is illustrated in the following example (which is an extract of code for converting λ-expressions to weak-head normal form from the experiment in Section 9.2).

```
(*[ whnf <: term -> whnf ]*)
(*[ apply <: whnf * term -> whnf ]*)

fun whnf (Var(n)) = Var(n)
  | whnf (Lam(e)) = Lam(e)
  | whnf (App(e1,e2)) = apply (whnf e1, e2)
and apply (Var(n), e2) = App(Var(n), e2)
  | apply (Lam(e), e2) = whnf (subst (e, Dot(e2, Shift(0))))
  | apply (App(e11, e12), e2) = App(App(e11, e12), e2)
```

Value sort specifications can also include comma-separated alternatives, which are treated just like alternative annotations for expressions.

7.2 Default sorts

One of the guiding principles in the design of our sort checker is that code which does not use sorts should be accepted without change in most cases. Thus, we would like to accept code fragments which contain no sort annotations at all. But, such code generally does not meet our grammar for checkable code, so it is not in a form suitable for direct checking by our bidirectional algorithm. We solve this by defining a default refinement for each type. When a required annotation is missing in an expression, the expression is treated as though it contains an annotation with the default refinement of the appropriate type.

The default refinements are chosen so that sort checking an unannotated term always succeeds provided that ordinary SML type checking succeeds and provided that there are no unmatched patterns in case statements. Most SML implementations give a warning when there are unmatched patterns, so all code which does not produce such a warning nor a type error will sort check without annotations. Thus, the default refinements in some sense "mirror" the ordinary SML types.

To make this more concrete, suppose we have the following datatype declarations.

```
datatype monotype = Int of int  |  Bool of bool
                  | Nil  |  Cons of monotype * monotype
                  | Fun of monotype -> monotype
```

Then the default refinement for `T` is generated by treating the program as though it contained a datasort declaration that mirrors the datatype declaration.

```
(*[  datasort monotype = Int of int  |  Bool of bool
                       | Nil  |  Cons of monotype * monotype
                       | Fun of monotype -> monotype
]*)
```

Here `int` and `bool` are the default refinements of the corresponding base types. These default refinements are the only refinements of these types, and they include all values of the type. The default refinement of a datatype always has the same name as the datatype. The default refinement of the type `monotype -> monotype` is the sort `monotype -> monotype`. In general, the default refinement is obtained by replacing each occurrence of a type name by its default refinement. This means that default refinements look exactly like the types that they refine.

These default sorts do not always include every expression of the corresponding type. E.g. The following expression is rejected by the sort checker because the pattern does not cover all of the sort `monotype`, and the default refinement is `monotype -> monotype`.

```
(* double has type: monotype -> monotype *)
fun double (Int x) = Int (x + x)
```

Such expressions with unmatched patterns are often an indication that there are interesting invariants that can be captured using sorts. Thus, it does not seem so bad that our sort checker rejects such code. In this case, a new datasort declaration could be added as follows.

```
(*[ datasort mtInt = Int of int ]*)

(*[ double <: mtInt -> mtInt ]*)
fun double (Int x) = Int (x + x)
```

Alternatively, code which is rejected can always be modified so that it is accepted by adding a catch-all case that raises an exception. For our example, this yields the following code.

```
fun double (Int x) = Int (x + x)
  | double _ = raise Match
```

For the convenience of those who choose to leave unmatched cases with respect to sorts, these errors are treated slightly differently from other errors. They are reported as "warnings", and appear before all other errors. Also, if the only errors encountered are such warnings, then the sort checking process is considered to have "succeeded".

Experience so far indicates that these default sorts are intuitive, convenient, and do not lead to unexpected results.

In some cases it is desirable to override the default sort with a different refinement. This is done by shadowing the default sort using a sort declaration that defines a sort with the same

name. This allows the programmer to in some sense "promote" a sort to a type, since all unannotated code will be checked against the default refinement chosen by the programmer. The following example demonstrates this mechanism.

```
(*[ sortdef monotype = mtInt ]*)

fun double (Int x) = Int (x + x)
```

This example sort checks because the type of `double` is `monotype -> monotype` and the default refinement of this type is `mtInt -> mtInt` due to the `sortdef` declaration.

7.3 Layered patterns

Standard ML includes one form of pattern that does not immediately fit into the scheme described in the Chapter 6. This is the "layered" form of pattern, i.e. the construct `x as` P which binds the variable `x` to the value that is matched against the pattern P.

The following example demonstrates the interaction between layered patterns and sort checking.

```
(*[ datasort ff = false
    datasort mtBool = Bool of bool
    datasort mtBaseList = Nil | Cons of mtInt * mtBaseList
                              | Cons of mtBool * mtBaseList
    and mtIntBaseList   = Nil | Cons of mtInt * mtBaseList
]*)

(*[ tailFrom <: (mtInt -> bool  &  mtBool -> ff)
                    -> mtBaseList -> mtIntBaseList   ]*)
fun tailFrom choose (wholeList as Cons (head, tail)) =
       if choose head then
            wholeList
       else
            tailFrom choose tail
  | tailFrom check Nil = Nil
```

The function `tailFrom` takes a monotype list containing integers and booleans, and returns the tail of the list starting from the first element accepted by the function `choose`. The sort assigned verifies that if `choose` always returns false for booleans, then the resulting list must start with an integer (or be empty).

When sort checking this example, the inversion principle for `mtBaseList` includes two cases for `Cons`, so we check the first clause of `tailFrom` against each. In the first case, the sort for `head` is `mtInt`, while in the second it is `mtBool`. To obtain the required the sort it is critical to assign the sort `mtIntBaseList` to the variable `wholeList` in the first case.

So, we cannot assign a sort to the variable in a layered pattern until after we have applied the inversion principles for the constructors in the pattern, and performed the resulting case analysis. Each case will assign sorts to the variables in the pattern, which we can then use to

synthesize a sort for the whole pattern using the sorts of the constructors in the pattern. The variable in the layered pattern is then assigned the synthesized sort.

This approach to assigning sorts to layered patterns is essentially the same as treating the variable `x` in a layered pattern `x as P` as if it was instead bound using `let` to an expression that follows the structure of the pattern `P`. For the example above, our approach is equivalent to transforming the layered pattern into the following code using `let`.

```
(*[ tailFrom <: (mtInt -> bool  &  mtBool -> ff)
                    -> mtBaseList -> mtIntBaseList   ]*)
fun tailFrom choose (Cons (head, tail)) =
     let val wholeList = Cons (head, tail)
     in
       if choose head then
            wholeList
       else
            tailFrom choose tail
     end
  | tailFrom check Nil = Nil
```

We additionally note that this approach to layered patterns is sometimes convenient when we would like to "improve" the sort of a variable that forms part of the object of `case` expression. For example, suppose we have the following function declaration.

```
(*[ f <: monotype -> mtInt ]*)
fun f x =
   case x of
      Int y => double x
    | _ => Int 0
```

This declaration results in a sort error, because the sort assigned to `x` is `monotype`, and `double` requires that its argument to have sort `mtInt`. But, inside the first branch of the case, we know that `x` matches `Int y` so we would like to assign `x` the more precise sort `mtInt`. We can do this by slightly changing the function declaration so that it introducing a new variable `x` that shadows the previous one, and is bound in a layered pattern, as follows.

```
(*[ f <: monotype -> mtInt ]*)
fun f x =
   case x of
      x as (Int y) => double x
    | _ => Int 0
```

An alternative to this might be to automatically "improve" the sorts of variables that appear in a case object, but we feel that this transformation is quite natural: if we require a more precise sort, we introduce a new variable so that we can assign it the more precise sort. Further, the presence of the layered pattern is likely to make it easier for the programmer to keep track of where such "improvements" are required.

7.4 Parameterized datatypes

Standard ML datatype declarations may include type parameters, such as `'a` in the following declaration.

```
datatype 'a option = NONE | SOME of 'a
```

This declaration defines a datatype constructor `option` which can be applied to any type. The development in Chapter 5 does not account for such parameters and type constructors, so in this section we consider the issues that arise when they are included.

7.4.1 Adding parameters to datasort declarations

Firstly, if we want to define refinements of these type constructors, then it seems natural to allow parameterized datasort declarations. We thus include such declarations with the restriction that the parameters must match the datatype declaration being refined, following the work of Freeman [Fre94] (we revisit this restriction later in this chapter). This allows us to define refinements of the above datatype with the following declarations (the first is the default refinement, and can be omitted).

```
(*[ datasort 'a option = NONE | SOME of 'a
         and 'a some = SOME of 'a
         and 'a none = NONE  ]*)
```

As in Chapter 5, datasort declarations such as these affect sort checking in three ways.

1. They determine the sorts for constructors.
2. They are used to calculate inversion principles.
3. They determine subsort inclusion relationships between datasorts.

The introduction of parameters in datasort declarations does not require any essential change in the way that sorts for constructors and inversion principles are determined. However, determining inclusions involving parameterized datasorts does require something new compared to Chapter 5. We describe our approach in the next subsection.

7.4.2 Inclusion of parameterized datasorts using variances

Our approach inclusion of parameterized datasorts is based on that used by Freeman [Fre94]. To determine inclusion between sorts constructed using parameterized datasorts we first compare the datasort bodies using the approach in Chapter 5 while keeping the datasort parameters abstract. We then compare the respective sorts to which the datasort constructors are applied, following the variance of each parameter. These variances are determined by the positions where each parameter occurs in the body of a datasort declaration.

For example, suppose we want to know whether the following inclusion holds.

$$\texttt{tt some} \leq \texttt{bool option}$$

First we compare the two datasort bodies, and find that the inclusion holds.

```
SOME of 'a ≤ NONE | SOME of 'a
```

Then, we compare the two sort arguments covariantly, since the parameter appears covariantly in the body of each datasort declaration. Thus, we check the following inclusion.

```
tt ≤ bool
```

This holds, and so we conclude that the original inclusion holds.

We differ from Freeman in that we allow each refinement of a datatype to have a different variance for its parameters. We also treat the interaction between intersections and covariant parameters differently from Freeman, to avoid including a form of distributivity.

Further, rather than infer the variance of parameters (as was done by Freeman) we provide a special syntax for declaring the variance of sort variables in the case when a datatype involves functions, references or other types that are not covariant in all positions. The four kinds of variance are: covariant, contravariant, ignored and mixed variance. The respective syntax for sort variables with these variances is as follows.

```
'+a     '-a     '?a     '!a
```

Making these variances explicit seems preferable, since it allows for better error messages when the actual variances differ from those intended by the programmer, and it allows weaker variances to be declared in anticipation of potential changes. It also allows the variances for abstract types to be declared in signatures, using the same syntax. Further, having variance information explicit in the syntax helps a programmer to understand the sorts associated with a program. In the common case of a datatype definition that doesn't involve functions, references, or non-covariant types, all parameters are assumed to be covariant unless otherwise specified.

Another difference from Freeman is that the underlying approach to inclusion of datasort bodies is based on Chapter 5 which is complete with respect to a straightforward inductive semantics (excluding functions), and so is arguably more intuitive and predictable. However, with parameterized datasort declarations it seems harder to directly characterize which inclusions hold in terms of inclusions of between sets of values. As pointed out by Skalka [Ska97], the approach of using variances to determine inclusions is fundamentally incomplete with respect to the the sets of values inhabiting each sort, even for the case of checking inclusion in an empty sort (i.e., checking emptiness). We discuss this in detail in the following subsection.

7.4.3 Incompleteness of inclusion via variances

The following example illustrates the incompleteness of our approach with respect to the sets of values inhabiting each sort.

```
(tt && ff) option ≤ bool none
```

Our approach rejects this inclusion because `'a option` is not included in `'a none`. However, when we consider the values inhabiting the two sorts, we find that they both contain exactly one value, namely `NONE`, so the values in the first sort are included in the second sort. The

reason the first sort only contains `NONE` is that any value of the form `SOME` V must have V inhabiting the sort `tt && ff`, but this sort is empty.

We can still characterize the subsortings as those that hold for all possible values of the sort parameters, but this does not directly extend to the case when sorts like `(tt && ff) option` and `bool none` appear in the datasort bodies, such as the following.

```
datatype aType = C of bool option

(*[ datasort aSort = C of (tt & ff) option
         and anEquivalentSort = C of bool none  ]*)
```

The work of Skalka [Ska97] resolves this difficulty, in the special case of checking emptiness of a sort, by substituting the sort arguments into a datasort body and then checking the emptiness of the resulting body, instead of checking the body while keeping the parameters abstract and then checking the arguments using variances. This substitution approach essentially reduces the problem back to that for the unparameterized case, and it seems that it should extend to checking inclusion rather than emptiness. A major technical accomplishment in the work of Skalka [Ska97] is a treatment of this substitution process that demonstrates that infinite loops cannot occur provided that the datatypes being refined do not use polymorphic recursion. The following example uses polymorphic recursion and would lead to an infinite loop when performing substitutions.

```
datatype 'a hyperList = Nil
                      | Cons of 'a * ('a hyperList) hyperList

(*[ datasort 'a even = Nil | Cons of 'a * ('a hyperList) odd
         and 'a odd = Cons of 'a * ('a hyperList) even
]*)


datatype 'a hyperTree = Leaf of 'a
                      | Node of  ('a hyperTree) hyperTree
                                 * ('a hyperTree) hyperTree

(*[ datasort 'a even = Leaf of 'a
                     | Cons of 'a * ('a hyperList) odd
         and 'a odd = Cons of 'a * ('a hyperList) even
]*)
```

Polymorphically recursive datatypes are rare in SML programs, perhaps because of the lack of polymorphic recursion in functions makes it impossible to deconstruct them recursively. So, perhaps it would not be so bad to forbid them completely in order to adopt the approach of Skalka. So far, we have chosen not to for a number of reasons.

1. One of the central goals of this work was to consider refinements in the context of a full, commonly used language. It would weaken our claim that we had done so if we started

introducing restrictions such as removing polymorphic recursion in datatypes. (Although this reason is perhaps less important than the others, particularly the next reason.)

2. Doing so greatly reduces the modularity of the analysis of datasort declarations. Using the variance approach, the analysis of a datasort declaration only depends upon minimal information about the sort constructors that appear in the datasort bodies. This information is the variance of parameters and the inclusions that hold between those sort constructors, and possibly also the inclusions between unions of sort constructors. Thus, using the variance approach makes it feasible to design a suitable extension of SML signatures that allow the specification of this information while keeping the datasort bodies opaque.

3. The substitution approach may have a serious impact on performance, because it requires inclusions for parameterized datasorts to be rechecked for each set of parameters. A sophisticated approach to memoization for inclusion results might avoid this by tracking the dependence of each result on the corresponding inclusions between the parameters.

4. Experience so far with the approach in the current implementation has not indicated a need for determining a richer set of inclusions.

We are hopeful that there is an extension of the variance approach that retains its modularity while matching the inclusions determined by the substitution approach. For the examples above involving the type `'a option` it would be sufficient to add to the variance information some "strictness" information that specifies that `'a option = 'a none` when `'a` is empty. In general though it seems that something more than this is required, as demonstrated by the following examples.

The first example demonstrates that the concept of strictness would need to be generalized to account for emptiness of multiple variables: the sort `option2` is contained in the `none` provided that *both* `'a` and `'b` are empty.

```
datatype ('a, 'b) option2 = NONE | SOME1 of 'a | SOME2 of 'b
(*[ datasort ('a, 'b) none = NONE  ]*)
```

The second example demonstrates that emptiness alone is not enough: information about unions is sometimes required also (and emptiness is then the zero-ary case of a union). In the declarations that follow, the same values are contained in both `aSort` and `anEquivalentSort`, but this fact depends upon the fact that `bool` is contained in the union of `tt` and `ff`.

```
(*[ datasort aSort = C of tt some | C of ff some
        and anEquivalentSort = C of bool some  ]*)
```

Consideration of these examples suggests that extending the variance approach to a form of *constrained inclusions* might obtain completeness with respect to inclusion of values. For the examples above, the appropriate constrained inclusions are the following.

$$\texttt{('a, 'b) option2} \leq \texttt{('c, 'd) none} \quad \text{when} \quad \texttt{'a} \leq \bot, \texttt{'b} \leq \bot$$

$$\texttt{'a some} \leq \texttt{('b some | 'c some)} \quad \text{when} \quad \texttt{'a} \leq \texttt{('b | 'c)}$$

The variance approach may seen as a special case of these constrained inclusions. It corresponds to constrained inclusions that only involve direct inclusions between the sort parameters of the respective datasorts, such as the following.

$$\texttt{'a some} \leq \texttt{'b some} \quad \text{when} \quad \texttt{'a} \leq \texttt{'b}$$

Consideration of the inclusion algorithm in Chapter 5 indicates that including constrained inclusions with constraints involving unions and emptiness should be sufficient to obtain completeness with respect to inclusions of values. Roughly, this is because such constraints are sufficient to express the "unsolvable" subgoals that would arise if the algorithm were applied "parametrically" to a goal involving some unknown sorts.

We intend to look into this further in future work, and we are also planning to experiment with the substitution approach.

7.4.4 Issues with instantiations of datatypes

Another issue related to parametric datasort declarations is that of directly defining refinements of particular instantiations of parameterized datatypes. For example, we might want to define a sort for lists of booleans which contain an even number of `true` values. We could try the following.

```
(*[ datasort evenTrues = nil
                       | :: of tt * oddTrues
                       | :: of ff * evenTrues
          and oddTrues = :: of tt * evenTrues
                       | :: of ff * oddTrues
]*)
```

Unfortunately, declarations like this must be rejected with the parametric approach that we currently use, because a datasort declaration is required to be parameterized in the same way as the datatype declaration that it refines. This leads to an awkward issue in practice: what do we do when we want to define such refinements of instantiations of pre-existing parameterized datatypes? It seems that the best that we can do with the current approach is to replace the instantiation of a parametrized datatype by a specialized datatype. For the `evenTrues` example above, this approach would add a new datatype for boolean lists, as follows.

```
datatype bools = bsnil
               | ::: of bool * bools
```

Then, the above datasort declarations for `evenTrues` and `oddTrues` can be easily modified to refine this type, by replacing occurrences of the constructors by `bsnil` and `:::`. The problem with this approach is that it prohibits the use of generic functions defined for the original parameterized datatype. Instead the code for such functions would need to be copied to the new instantiated datatype, leading to code maintenance problems.

This issue seems like one that would arise reasonably often in practice. In fact, it arose in one of the examples described in Chapter 9. In that example, part of the code for a parser used

a list to represent a stack of unresolved operators, with some relatively complicated invariants concerning where different sorts of operators could occur in the list. This was unproblematic for this example, because none of the standard list functions were used in the code. However, it seems highly likely that situations would arise in practice where refinements of instantiations of datatypes are desirable, but the generic functions for these datatypes need to be used extensively.

Thus, it seems desirable to find some way to allow such refinements to be defined without introducing a completely new type. It appears that this would fit better with the substitution approach to parameterized datasort inclusion than the variance approach, since with the substitution approach we could directly compare datasort definitions like `evenTrues` with substituted instances such as `tt list`.

However, it does seem possible to extend the variance approach so that it allows datasort declarations that are still parameterized, but which have instances that correspond to sorts like `evenTrues`. To achieve this, we need to allow more than one parameter refining the corresponding type parameter in the datatype declaration. For example, this would allow us to define `evenTrues` with the following declarations.

```
(*[ datasort ('a, 'b) even1List = nil
                                | :: of 'a * ('a, 'b) odd1List
                                | :: of 'b * ('a, 'b) even1List

          and ('a, 'b) odd1List =  :: of 'a * ('a, 'b) even1List
                                | :: of 'b * ('a, 'b) odd1List

    sortdef evenTrues = (tt, ff) even1List
]*)
```

While this design seems to be a promising solution, it is not at all clear what is the best way to resolve these issues. Some more experience with programming with the current implementation will help to judge how often these issues arise, and what solution is most appropriate in practice. Regardless, this seems like an interesting and potentially fruitful direction for future research.

7.5 Parametric polymorphism

Standard ML includes type schemes with parametric polymorphism, unlike the formal languages considered in prior chapters. This is relatively major difference, and in fact parametric polymorphism is generally considered one of the defining features of an ML-like language.

Following Freeman [Fre94], we define the refinements of polymorphic type schemes as corresponding sort schemes, with the restriction that there is exactly one sort variable refining each type variable. This allows our approach to extend to languages with parametric polymorphism, although we still need some method for instantiating the sort schemes of polymorphic variables.

When we have an occurrence of a variable with a polymorphic type scheme, ordinary ML type inference will determine an appropriate instantiation of the type scheme. The corresponding sort scheme can be instantiated with any refinement of this type, and thus the sort scheme can be instantiated by enumerating the distinct refinements of the type. This is the basic approach used by our implementation of a sort checker for SML.

Alas, the number of distinct refinements of a type can be huge, as discussed in Section 3.10, and so in some situations this enumeration is far from practical. This is particular true when the type scheme of a polymorphic variable is instantiated with a function type that has a non-trivial lattice of refinements. Such instantiations seem to be reasonably rare, although it is clear that they do occur in practice. In such situations, the current design requires the occurrence of the polymorphic variable to appear in a checkable position in the program. This may require the programmer to annotate the variable with the intended sort instance.

An earlier version of the implementation did not enforce this restriction: instead violations of it potentially lead to infeasible enumerations. This seems unsatisfactory, so we have recently made this a strict requirement: such variable occurrences are assigned only the default refinement if they appear in non-checkable positions. This generally avoids impractical instantiations, and leads to appropriate error messages when required annotations are absent.

However, it seems possible that there are situations where the annotations required by the current approach are excessive, and more experience is needed to determine whether the current solution is sufficient in practice. In only one of our experiments so far have we encountered variable occurrences that required annotations. In that case the required annotations seemed reasonable, although only after we defined new variables for particular instantiations of the polymorphic variable that were used repeatedly (this will be discussed further in Section 9.3).

If further experience indicates that our current approach leads to excessive annotations, it seems possible that many annotations could be avoided using an approach similar to the local type inference of Pierce and Turner [PT98]. However, adapting local type inference to our situation appears to be non-trivial, in particular due to the presence of intersections.

A very promising and simpler alternative is to allow the programmer to provide "mode" annotations for the problematic variables, which specify appropriate information about how a sort scheme should be instantiated based on sorts associated with the context of an occurrence. For example, it might be specified that a sort scheme for a function should be instantiated by inferring the sort for the argument, thus instantiating via the argument sort of the function. Or, it might be specified that applications of the function must appear in checkable positions, with the sort scheme instantiated via the result sort of the function.

This appears to be an appropriate extension of bidirectional checking to polymorphic variables: in particular it allows polymorphic variables to be defined for which sort checking proceeds similarly to built-in constructs. For example, the following signature specifies a type `bits` for bit strings with refinements `nat` and `pos`, along with values for constructors `bnil`, `b0` and `b1` and with a function for a single-level "case" construct.

```
    type bits               (*[ sortdef nat < bits
                                    and pos < nat ]*)

    val bnil : bits         (*[ val bnil <: nat ]*)
    val b0 : bits -> bits   (*[ val b0 <: pos -> pos ]*)
    val b1 : bits -> bits   (*[ val b1 <: nat -> pos ]*)

    val bcase  : bits -> (unit -> 'a) -> (bits -> 'a) -> (bits -> 'a) -> 'a
(*[ val bcase <: bits -> (unit -> 'a) -> (bits -> 'a) -> (bits -> 'a) -> 'a
              & nat -> (unit -> 'a) -> (pos -> 'a) -> (nat -> 'a) -> 'a
              & pos -> (unit -> 'a) -> (pos -> 'a) -> (nat -> 'a) -> 'a   ]*)
```

Then, we could specify that applications of this function should only appear in checkable positions, that the first argument should be inferable and the second third and fourth arguments checkable, with a declaration along the lines of the following.

```
(*[ mode bcase : inf -> chk -> chk -> chk -> chk ]*)
```

In this case it is clear that the sort variable `'a` should be instantiated via the sort that the whole application is checked against. This is because the occurrence of `'a` in this part of the sort for `bcase` is in an "input" position, i.e., this sort will be known each time an application of `bnat` is checked, while all other occurrences of `a` appear within sorts of checkable arguments, which are "output" positions, i.e., sorts that need to generated each time an application of `bnat` is encountered.

This approach requires each sort variable in a sort scheme to appear in at least once in an input position. It is not entirely clear what to do when a sort variable appears multiple times in input positions: this could be disallowed, or the resulting instantiating sort for each occurrence could be tried in sequence. Alternatively, some precedence could be defined so that e.g., occurrences in earlier arguments are used for instantiation in preference to later arguments (resulting in subsorting checks against the sorts obtained from other occurrences). Yet another approach would be to form upper and lower bounds as appropriate according to the variances of the positions where the sort variable occurs, although this might not fit well with our implementation in the case of upper bounds, which can be very expensive to compute.

A more flexible approach would be to allow the occurrence (or occurrences) that should be used for instantiation to be designated by the programmer. For example, one way this might be done is illustrated in the following specification for a polymorphic function which randomly decides whether to swap the two components of a pair. The sort of the argument is repeated in the mode specification, and the second occurrence in the argument sort is designated by modifying the sort variable to the form `'^a`.

```
val maybe_swap : 'a * 'a -> 'a * 'a   (* Swaps with prob. of 0.5. *)

(*[ mode maybe_swap : inf['a * '^a] -> inf ]*)
```

There appear to be a number of possible choices here as to exactly how to mode information should be expressed, and what should be allowed. A full investigation of these choices is left to future work, but it seems reasonably clear that the basic concept of allowing such modes to be declared for polymorphic variables is more satisfactory than the approach to parametric polymorphism that is currently included in the implementation.

7.6 Other core SML constructs

Extending to the rest of the core language of Standard ML is relatively straightforward, and our implementation allows all Standard ML programs to be checked. Some SML features present interesting opportunities to exploit refinements, such as the extensible `exception` datatype. We have not attempted to pursue such opportunities, but plan to in future work.

We briefly comment on the few remaining interesting features of our extension.

7.6.1 Exceptions

The extensible nature of the exception type, and the relationship between exceptions and control flow present some potentially interesting opportunities and challenges for the use of sorts. We have not yet properly investigated these issues. Our design for exceptions is relatively basic, and is designed to extend sort checking to programs involving exceptions in a convenient way.

For each declaration of an SML exception, we assign the exception constructor the default refinement of the appropriate type. There is currently no way to declare another refinement for an exception constructor.

When an exception is matched in a case statement, we treat the exception type as "open ended", in the sense that even if every defined exception is listed, it is still considered that there are unmatched cases. This is because the exception type is extensible.

In a `handle` expression, we do not consider it a sort error for there to be unmatched cases. So, the issue of open-endedness does not result in sort errors in this common case of matching against exceptions. Allowing such unmatched cases is consistent with the normal use of such expressions.

Expressions of the form `raise E` are checkable only, not inferable, since it is in many cases it is expensive to construct the principal sort, which is the least refinement of the corresponding type. Similarly `handle` expressions are checkable only, to avoid the need to take an upper bound of sorts of the unexceptional expression and the exceptional expressions.

In future work, we may consider allowing refinements to be declared using a syntax like the following.

```
exception myException of bool * bool
(*[ exception myException of tt * bool
          | myException of bool * tt   ]*)
```

Such a declaration would declare a sort and an inversion principle for `myException` other than the default.

We might additionally consider the definition of refinements of the exception type itself.

```
(*[  datasort allowedExceptions = myException of bool * bool
                                | anotherException  ]*)
```

One issue that arises here is whether we should allow refinements of the exception datatype that are themselves extensible.

Another potential future direction is to consider refinements that track the potential exceptions raised by expressions, which seems to fit within the framework for refinements for effects described by Mandelbaum, Walker and Harper [MWH03].

7.6.2 Let expressions

SML expressions include a let form which may introduce a local set of core-level declarations for an expression. Such expressions are inferable if the body of the let is, otherwise they are checkable. Any value declarations introduced should involve expressions which are inferable (perhaps because they contain a sort constraint) otherwise the default sort will be used, as described in Section 7.2.

7.6.3 Local datatype and datasort declarations

Local datatypes in SML may in some sense "escape" from the scope of their declaration, as in the following example.

```
local
  datatype myBool = myTrue | myFalse
in
  fun myNot myTrue = myFalse
    | myNot myFalse = myTrue
  val myBoolVal = myTrue
end
val myResult = myNot myBoolVal
```

Local datasort declarations are treated in the same way, so the following is allowed.

```
local
  datatype myBool = myTrue | myFalse
  (*[ datasort myTT = myTrue  and  myFF = myFalse ]*)
in
  (*[ myNot <:  myBool -> myBool  &  myFF -> myTT ]*)
  fun myNot myTrue = myFalse
    | myNot myFalse = myTrue

  (*[ myBoolVal <: myTT ]*)
  val myBoolVal = myTrue
end
val myResult = myNot myBoolVal
```

Notice that in this example, the inclusion of `myTT` in `myBoolVal` is required in order to check the application of `myNot` to `myBoolVal`. Thus, information about inclusion relationships needs to be retained after the scope of a datasort declaration, and needs to be associated with sorts that depend on the datasort declaration.

Following SML, we use the notion of type names and introduce the corresponding notion of sort names to track this information. During sort checking we have also have a *type-name refinement environment*, which maps each type name to the required information about the sortnames that refine it. These type-name environments play a similar role to the sets of type names in the static semantics of Standard ML [MTHM97], except that some extra information is associated with each type name, so they are environments rather than sets.

One effect of this design is that the inversion principles and sorts of constructors that are affected by a datasort declaration do not revert when leaving the scope of the datasort declaration. This seems consistent with the SML treatment of datatypes. On the other hand, in some cases it does seem desirable to be able to make local datasort declarations without this affecting the sort checking of other parts of the program. We intend to look into this in future work.

7.6.4 Patterns containing `ref`

SML pattern matching can be used to match against a reference cell. The following example demonstrates that this can leads to unsoundness unless we treat reference cells differently when sort checking instances of pattern matching.

```
(*[ datasort tt = true and ff = false ]*)

val r = ref true
fun set () = r := false

(*[ f <: bool ref -> tt ]*)
fun f (y as (ref true)) = (set(); !y)
  | f (ref false) = true

(*[ x <: tt ]*)
val x = f r
```

In this example, if we treat `ref` like other constructors, then the sort assigned to `y` will be `tt ref`. But, the call the `set` modifies the reference so that `!y` returns `false`.

To remedy this, we always determine the sorts for such variables using the original sorts assigned to the reference, i.e. the sorts assigned in the input to the first branch. This does not affect the use of pattern sort subtraction to determine reachability of each branch: it is sound to treat `ref` like other constructors when subtracting, as long as the result of the subtraction is not used to determine the sorts of variables.

7.6.5 Datatype replication

When an SML datatype replication expression like the following is encountered, the default refinement is replicated along with the datatype.

```
datatype myOption = datatype option
```

There is no need for a corresponding datasort replication construct. The required refinements can simply be replicated using `sortdef`, as in the following example.

```
(*[ sortdef 'a mySome = 'a some
        and 'a myNone = 'a none ]*)
```

Datasort can also be replicated using datasort declarations which achieves the same result.

7.6.6 Boolean conditionals

SML includes `if` boolean conditional expressions like the following.

```
if done then
    finish ()
else
    keep_going ()
```

The definition of SML [MTHM97] defines such expressions to be abbreviations for a corresponding `case` expression. The example above is thus an abbreviation for the following.

```
case done of
   true => finish ()
 | false => keep_going ()
```

This turns out to be convenient, since it means that the appropriate inversion principle for `bool` is applied. For example, the example at the end of Section 7.3 requires that the inversion principle for `ff` be used so that only the false branch is checked.

7.7 Modules

Extending sort checking to include the module system of SML presents a few new issues. We have focused on relatively minimal design that allows all SML programs to be checked. Thus, to large extent the design for sorts follows the corresponding treatment of types in the Definition of Standard ML [MTHM97]. There is certainly scope to consider a richer design, and we expect experience with the current design will help in determining what additional features are desirable.

7.7.1 Structures

SML structures include declarations from the "core" language, i.e. the non-module declarations. The sort annotations we have considered up to now are an addition to that core language, and hence they can appear as part of structures. Thus, structures can include `datasort` declarations and `sortdef` declarations. They can also include sort annotations on expressions, and value sort specifications. The following is an example of the use of these forms in a structure.

```
structure Monotype =
struct
   datatype monotype =
       Int of int  |  Bool of bool
     | Nil  |  Cons of monotype * monotype
     | Fun of monotype -> monotype

   (*[ datasort mtInt = Int of int ]*)

   type mtArrow = monotype -> monotype
   (*[ sortdef mtIntToInt = mtInt -> mtInt ]*)
   (*[ sortdef mtAllToInt = monotype -> mtInt ]*)

   (*[ val double <: mtIntToInt ]*)
   fun double (Int x) = Int (x + x)

   (*[ val tryDouble <: mtAllToInt ]*)
   fun tryDouble (Int x) = double (Int x)
     | tryDouble nonInt = raise Match

   val quiteLikelyFour = double (Int 2 (*[ <: mtInt ]*) )
end
```

One minor difference compared to the situation without structures is that `datasort` declarations can include constructor names that include a path to the structure containing the corresponding datatype declaration, as in the following example.

```
structure mtBaseList =
struct
  (*[ datasort mtIntList =
         Monotype.Nil
       | Monotype.Cons of Monotype.mtInt * mtIntList
  ]*)

  (*[ append <: mtIntList -> mtIntList
                          -> mtIntList ]*)
  fun append (Monotype.Cons (h, t)) l =
        Monotype.Cons (h, append t l)
    | append Monotype.Nil l = l
end
```

7.7.2 Signatures

Extending the signatures of SML to include sorts requires a little more thought than structures, in particular for refinements of opaque types. We have datasort, sort, subsorting, and value sort specifications in signatures, as in the following example, which is a signature that could be assigned to the structure `Monotype` above.

```
signature MONOTYPE =
   sig
     datatype monotype =
         Int of int  |  Bool of bool
       | Nil  |  Cons of monotype * monotype
       | Fun of monotype -> monotype

     (*[ datasort mtInt = Int of int ]*)

     type mtArrow
     (*[ sortdef mtIntToInt |: mtArrow ]*)
     (*[ sortdef mtAllToInt < mtArrow ]*)

     (*[ subsort mtAllToInt < mtIntToInt ]*)

     (*[ val double <: mtInt -> mtInt ]*)
     val double : monotype -> monotype

     (*[ val tryDouble <: mtAllToInt ]*)
     val tryDouble : mtArrow

     (*[ val quiteLikelyFour <: mtInt ]*)
     val quiteLikelyFour : monotype
  end
```

Datasort specifications in signatures are transparent, in the sense that they expose the datasort declaration that must appear in any structure matching the signature. We have the restriction

that the datasort must be a refinement of a datatype that is itself transparent. In the example, `mtInt` is a datasort specification refining the transparent datatype specification `monotype`.

Value sort specifications specify sorts for values which have their types specified in same signatures using ordinary SML value specifications. The specified sort for a value must be a refinement of the corresponding type. In the examples, the sorts declared for `double` and `quiteLikelyFour` are both refinements of the corresponding types. Comma separated alternatives are not allowed in signatures: there is no need for them, since we never need to backtrack over module level declarations.

Sort specifications specify refinements of opaque types. In the example above, `mtIntToInt` is specified to be a refinement of the opaque type `mtArrow` (the symbol `|:` is intended to represent $\sqsubset$). Similarly, `mtAllToInt` is specified to be refinement of the same type that must be a subsort of the default refinement of `mtArrow`, using and alternative form of sort specification allows an upper bound to be specified. This form seems to be convenient in practice, although the upper bound could instead be specified using a separate subsorting specification.

Subsorting specifications specify inclusions between refinements of opaque types. In the example above, the refinement `mtAllToInt` is specified to be a subsort of `mtIntToInt`. Subsorting specifications have some similarities with the type sharing specifications of SML. In particular, the syntax for a subsorting specification includes the whole specification that precedes it, and modifies the environment of that specification by equating some sorts. For example, the subsorting declaration above results in the sorts `mtAllToInt` and `mtAllToInt & mtIntToInt` being equal. This form of subsorting specification allows any finite lattice of refinements to be declared, just like the the subsorting declarations in signatures in Chapter 2.

One weakness of our design with respect to refinements of opaque types is that currently they are not allowed for types that are later instantiated using `where`. It seems that what is required to allow such refinements is a corresponding construct that instantiates them. For this to work, when a type is instantiated using `where`, we must require that all refinements are instantiated appropriately. For now, when refinements of such types are needed, the signature needs to be copied and instantiated manually in order to make the types transparent without using `where`.

One of the key features of the SML modules system is the ability to specify sharing between types. In our current design, all refinements of types that are involved in a sharing specification are matched up by name, and must correspond precisely between the two types. This is convenient in the common case where all refinements of a type are defined in the same module as the type, and propagate though modules via various paths along with the type. In this case the type sharing declarations automatically force the desired sharing of refinements. However, it seems that a more general mechanism which allowed separate specification of sort sharing might be desirable in some cases.

Variance annotations are are particularly important for opaque sorts in signatures. They provide a mechanism for specifying partial information about the sorts that must appear in any structure matching a signature. This allows client code to make use of the variance information during sort checking, without depending on the exact implementation of a sort in a particular structure implementing the signature.

Datatype replication in signatures also replicates the default sort, just as in structures. Similarly, no datasort replication mechanism is needed in signatures, instead datasort declarations can be used since datasort declarations are not generative.

One additional feature is included in the modules language of sorts. It allows the programmer to specify that the sort checker should assume that a structure has a particular signature without actually checking the contents of the structure at all. This is used in the implementation to skip checking of all the structures in the SML Standard Basis each time the system is started. It seems that it might also be useful when checking large systems in the case that some modules are unlikely to change and have already been checked, or when some modules have been determined to satisfy the specified sorts in some way other than by running the sort checker.

The following is an example of the use of this construct, which is taken from the stub code used to set up the sorts for the Standard Basis. (Currently no interesting refinements are defined in these stubs, but they are necessary to allow checking code that uses the Standard Basis.)

```
structure Bool:BOOL =
struct (*[ assumesig BOOL ]*)

     type bool = bool
     fun not true = false
       | not false = true

     fun toString false = "false"
       | toString true  = "true"

     fun getstring str getc source =
     (* ... etc. (remaining declarations omitted) *)
end
```

Here, the use of `assumesig` specifies that the sort checker should skip checking of the declarations in the body of the structure, and instead assume that they satisfy the signature `BOOL`.

7.7.3 Signature matching

Matching a structures against signatures is done in essentially the same way at the level of sorts as at the level of types. One difference is that the sort schemes for values in a structure may be subsorts of the corresponding sorts in the signature. As for types, the sort schemes in the structure may be more polymorphic than those in the signature. If this is the case, the sort scheme in the signature must match the form of the sort scheme in the structure. This is because the combination of instantiation and subsorting is expensive to check in general, as discussed in Chapter 7.5.

Also, we allow the declared variance of sort constructors to be richer in the structure than the signature.

When matching against an lattice of refinements of an opaque type, we check that the sorts realizing the refinements satisfy the specified lattice structure, i.e. that every specified inclusion is satisfied (and perhaps more).

7.7.4 Functors

There are no particularly new issues that arise when we consider sorts for functors. Essentially, we have covered all the issues in the sections on signatures and structures.

7.8 Grammar for the language with sort annotations

For completeness, we now present a formal grammar in the style of the Definition of Standard ML [MTHM97] for our extended language including sort annotations. We do not separate out inferable and checkable expressions here, since the implementation does not have a restriction on the form of expressions allowed. Instead, it reverts to using the default refinement when required annotations are absent, as described in Section 7.2.

sort ::= *tyvar*
| `{` ⟨*sortrow*⟩ `}`
| *sortseq longtycon*
| *sort*$_1$ `->` *sort*$_2$
| `(` *sort* `)`
| *sort*$_1$ `&` *sort*$_2$
| `topsort[`*type*`]`

sortrow ::= *lab* `<:` *sort* ⟨ `,` *sortrow*⟩

sortlist ::= *sort* ⟨`,` *sortlist*⟩

pat ::= ...
| *pat* `(*[` `<:` *sortlist* `]*)`

atexp ::= ...
| `(` *exp* `(*[` `<:` *sortlist* `]*)` `)`

dec ::= ...
| `(*[` *sortdec* `]*)`
| `(*[` *valsortdec* `]*)`

sortdec ::= `val` *valsortdec*
| `datasort` *datsortbind* ⟨`withsort` *sortbind*⟩
| `sortdef` *sortbind*
| *sortdec*$_1$ ⟨`;`⟩ *sortdec*$_2$

valsortdec ::= *vid* `<:` *sortlist* ⟨`and` *valsortdec*⟩

datsortbind ::= *tyvarseq tycon* `=` *consortbind* ⟨`and` *datsortbind*⟩

consortbind ::= ⟨`op`⟩ *longvid* ⟨`of` *ty*⟩ ⟨`|` *consortbind*⟩

sortbind ::= *tyvarseq tycon* `=` *sort* ⟨`and` *sortbind*⟩

$$
\begin{array}{rcl}
\mathit{spec} & ::= & \ldots \\
& | & \texttt{(*[}\ \mathit{sortspec}\ \texttt{]*)} \\
& | & \mathit{spec}\ \texttt{(*[subsort}\ \mathit{longinter}_1\ \texttt{<}\ \mathit{longinter}_2\ \texttt{]*)} \\
\\
\mathit{sortspec} & ::= & \texttt{val}\ \mathit{valsortspec} \\
& | & \texttt{datasort}\ \mathit{datsortspec} \\
& | & \texttt{sortdef}\ \mathit{sortspec} \\
& | & \mathit{sortspec}_1\ \langle\texttt{;}\rangle\ \mathit{sortspec}_2 \\
\\
\mathit{valsortspec} & ::= & \mathit{vid}\ \texttt{<:}\ \mathit{sort}\ \ \langle\texttt{and}\ \ \mathit{valsortspec}\rangle \\
\\
\mathit{datsortspec} & ::= & \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{consortspec}\ \ \langle\texttt{and}\ \ \mathit{datsortspec}\rangle \\
\\
\mathit{consortspec} & ::= & \mathit{longvid}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \ \langle\texttt{|}\ \ \mathit{consortspec}\rangle \\
\\
\mathit{sortspec} & ::= & \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{|:}\ \mathit{longtyconinter}\ \ \langle\texttt{and}\ \ \mathit{sortspec}\rangle \\
& | & \mathit{tyvarseq}\ \mathit{tycon}\ \ \texttt{<}\ \ \mathit{longtyconinter}\ \ \langle\texttt{and}\ \ \mathit{sortspec}\rangle \\
\\
\mathit{longinter} & ::= & \mathit{longtycon}\ \langle\ \texttt{\&}\ \ \mathit{longinter}\rangle \\
\\
\mathit{strexp} & ::= & \texttt{struct (*[assumesig}\ \mathit{sigexp}\ \texttt{]*)}\ \mathit{strdec}\ \texttt{end}
\end{array}
$$

Currently there is no direct syntax for specifying empty datasorts: the only way to refer to them is via intersections of disjoint refinements of a datatype.

Chapter 8

Implementation

We have built an implementation of sort checking for Standard ML based on the ideas in the preceding chapters. In the context of this thesis work, this implementation demonstrates that our approach can be scaled up to a fully featured language such as SML. It also allows us to evaluate our language design and algorithms by performing realistic experiments with programming with sorts using real SML programs, as reported in Chapter 9.

8.1 Reuse of an existing SML front end

In order to check SML programs, our sort checker needs to include a parser for SML. Additionally, since our sort checker assumes that its input has already been type checked, and requires type information produced during type checking, we require a type checker for SML. We could have build our own parser and type checker for SML, but this would have been a relatively large amount of work.

Thus, we have chosen to reuse the front end of an existing SML compiler, namely the ML Kit compiler, version 3. The ML Kit compiler was chosen because it was designed to be easy to modify and extend. It is also particularly suitable because it includes a mechanism that adds annotations to the abstract syntax tree, including the typing information required by the sort checker.

8.1.1 Integration with the ML Kit elaborator

The Kit compiler implements the elaboration phase of SML in a way that closely follows the definition of SML [MTHM97]. The elaboration phase produces an abstract syntax tree that contains annotations with a variety of information, including type information. We use this annotated abstract syntax tree as the input to the sort checking phase.

We avoid commencing the sort checking phase when there is a type error found during elaboration. This allows the sort checker to depend upon the type correctness of the input, which avoids the consideration of many error cases that should already be caught during elaboration.

The new syntax for sort annotations was added to the parsing code of the ML Kit. The elaborator was modified so that it also checks that the types associated with these annotations annotations are consistent. This means that the sort checker can always rely on types being consistent, even in the sort annotations themselves.

At the module level it is convenient to integrate sort checking directly into the elaborator, since a separate sort checking phase would have to duplicate much of the code the elaborator in order to reproduce the environments used during elaboration. Thus, our design adds the code for sort checking directly to the elaboration code for each module level construct. This sort checking code is only run when no elaboration errors have been encountered so far in any part of the program. When a core level object is encountered (such as the body of a structure) the elaboration code for the object is called, and if no errors are discovered the sort checking code for the object is called.

8.1.2 An alternative approach to integration

The core level sort checking code also requires a variety of information from the corresponding code in ML Kit elaboration phase. It obtains this information via the annotations on the AST produced by the elaboration phase, which worked quite well in most cases. However, in some cases the code in the sort checker is quite tightly coupled with the corresponding code in the elaboration, and communicating via the AST is somewhat clumsy. In addition, the core level sort checking code also needs to duplicate some of the work done in the elaboration phase to determine the environments and types used during elaboration. Much of the time spent debugging was related to bugs where the sort checker somehow came to use environments and sorts that were inconsistent with those used during elaboration.

We could attempt to avoid these issues by integrating the sort checking phase into the elaboration code for each construct, as is done at the module level. However, this is not so easy, because the sort checking phase requires that type inference has been completed. In particularly, the use of side effects when unifying types means that the type of each expression may not be known until elaboration of a whole core-level object has completed. One way around this would be to use higher order functions to separate the two stages, so that the elaboration code for each construct returns a result that includes a function for performing the required sort checking. This would be an interesting design to pursue, and would allow the sort checking code access to all the relevant data structures used during elaboration without the need to explicitly store those structures in the abstract syntax tree.

Such an approach may be worth considering in potential future implementations of refinement checking. The main reason this approach was not used in this work was that it seemed important for modularity to separate the sort checking code as much as possible from the elaborator. Also, the possibility of using higher-order functions in this way was only realized after a reasonably large commitment had been made to the current design.

8.2 Representations: types, sorts and environments

Since our sort sort checking code depends upon the elaborator in the ML Kit, we use the the ML Kit representations of the static objects of SML, i.e. types, type variables, type names, etc. We also use of some of the representations of environments used by the elaborator. These representations were also used as a starting point for the representations of the corresponding notions at the level of refinements, i.e. sorts, sort variables, sort schemes, sort functions, sort names, etc. and the various environments used during sort checking.

We could have instead decided to modify the existing representations at the level of types so that they could be used at the level of sorts as well. For example, we could have added intersections to the representation of types instead of creating a new representation for sorts. This would have avoided the need to duplicate of a lot of code for basic operations related to types. It would also have allowed the reuse of the representation of environments used by the elaborator, thus would similarly have avoided the need for duplication of basic operations related to environments.

The choice was made to have a separate representation for sorts from types for a number of reasons.

1. The representation of types in the ML Kit is much more complicated than what is needed during sort checking. In particular, the representation of types is specifically designed to allow unification by imperative update, while keeping track of "binding levels" of types following the type inference algorithm of Rémy[Rém92].

2. Some of the basic operations at the level of types are clearly not implemented in the way that is appropriate for our sort checker. For example, applying a substitution to a type is implemented by code that essentially just returns the type unchanged. This is because these "substitutions" are an artifact of a previous version of the ML Kit that did not use imperative update for unification but instead accumulated substitutions. In the current version, all substitutions are done "on the fly". However, our sort checker is not based on unification at all, and we certainly need a proper implementation of substitution.

3. Our environments are different enough from those at the level of types that it would have been awkward to extend the ML Kit environments in a way that was suitable for both elaboration and sort checking. We could have attempted to use a functor to parameterize the common parts of the implementation with respect to those parts that are different, but this would still likely have been awkward, due to the mutual recursion between the main data types used to represent environments.

4. Using different representations results in greater modularity between the existing ML Kit code and that added for sort checking. This reduces the chances that our additions will result in bugs in the elaborator. We did uncover a number of such bugs, and were able to quickly assess that they could not have been introduced by us. This modularity also allows the modifications made in updated versions of the ML Kit elaborator to be incorporated relatively easily, which is desirable since these versions may fix important bugs.

8.3 Representation of intersections in sorts

One higher level choice that we have made in the implementation is in the way that we handle intersections in sorts. In particular, we "simplify" all sorts in the manner of Section 3.7,so that intersections only appear between refinements of function types, reference types, and parameterized data types that involve either function or reference types. We do this for efficiency, and also because it reduces the number of cases that need to be considered when analyzing sorts.

For refinements of record types, we distribute intersections inwards, using the following equivalence, as in Chapter 6.

$$(R_1 \times R_2) \,\&\, (S_1 \times S_2) = (R_1 \,\&\, S_1) \times (R_2 \,\&\, S_2)$$

For refinements of data types and opaque types, we introduce a new *sort name* to represent each unique refinement that can be constructed using intersections. These sort names are the internal identifiers for sorts, just like the type names in the used in definition of SML for datatypes and opaque types. The implementation constructs the lattice of refinements for each type name based on upon the declarations of the refinements, and stores the intersection for each pair of refinements.

With these sort names, every refinement of a type constructed from a type name can usually be represented as a simplified sort without intersections. The exception to this is when we have parameterized types with non-covariant parameters, since in this case we cannot combine the parameters of the different parts of the intersection. For example, we cannot simplify the intersection in the following.

```
datatype ('a, 'b) function = Function of 'a -> 'b

(*[ val myFunc <: (tt, ff) function & (ff, tt) function ]*)
val notFunct = Function (fn x => if x then false else true)
```

In such cases, we still use a different sort name to represent each unique refinement of the parameterized type, since we depend on having a different sortname for each refinement in many parts of our implementation.

We represent the operation $r \,\&\, s$ for each lattice as a map from a pair of sort names to a sort name for the intersection. We omit the reflexive intersections $r \,\&\, r$, and we avoid storing both $r \,\&\, s$ and $s \,\&\, r$ by using the order on the integers that are used as internal identifiers for sortnames. Currently the data structure used to implement these maps is based on an implementation of red-black trees by Pfenning that is used in the Twelf implementation [PS98]. During sort checking, these maps are stored in *type name refinement environments* which store information about the refinements of each type name, and which play a similar role to the type name sets in the Definition of Standard ML [MTHM97].

8.4 Analysis of datasort declarations

When a `datasort` declaration is encountered, we need to determine three things: a modified lattice of refinements, the inversion principles, and the modified sorts for constructors.

To determine the modified lattice, the body of each new datasort declaration is checked against each of the current lattice elements that refine the same datatype, checking for equivalence using an implementation based on the datasort inclusion algorithm in Chapter 5 (see below). If an equivalent element is found, the lattice is unchanged, and the datasort declaration binds a new name to the existing element. Otherwise a new element is added to the lattice, and then we continue by constructing the intersection with every other element in the lattice and then adding a new lattice element for the intersection whenever it is not equivalent to an

existing lattice element. At the end of this process all intersections of the new element with another have been added to the lattice.

To complete the lattice, we still need to add the intersections of the new intersection elements with other lattice elements. Fortunately, we never need to add any further lattice elements: the lattice element representing the intersection between ρ and ρ_{new} & ρ_{old} is determined using associativity, as follows.

$$\rho \ \& \ (\rho_{\text{new}} \ \& \ \rho_{\text{old}}) = (\rho \ \& \ \rho_{\text{new}}) \ \& \ \rho_{\text{old}}$$

We have then finished calculating the lattice that results from the datasort declarations. We continue by determining the inversion principles and sorts of constructors following Chapter 5, removing redundant sorts in each case. We have then finished the analysis of the datasort declaration.

The only remaining part of the implementation of analysis of `datasort` declarations is that which determines equivalence between datasorts. We do this by checking inclusion in each direction: there seems to be no obvious way to obtain a more efficient algorithm for the equivalence problem (the problematic case is when we have a union of products). The code for determining datasort inclusion is one of the most complicated parts of the implementation. It is also the part that has required the most optimization to obtain acceptable performance.

Our implementation of inclusion between datasorts is based on the algorithm in Chapter 5, extended with variances as in Chapter 7. This algorithm may take exponential time, and we cannot expect to obtain any better worst case performance: the problem of inclusion for regular tree languages is EXPTIME-complete, as shown by Seidl [Sei90]. However, some evidence indicates satisfactory performance is possible when the instances of the inclusion problem are based on the use of regular tree grammars to describe sets of values in a programming language. This evidence includes the work of Aiken and Murphy [AM91], and more recently the work of Hosoya, Vouillon and Pierce [HVP00] and of Benzaken, Castagna, and Frisch[BCF03]. In each of these works, some optimizations were found to be critical in obtaining acceptable performance, and our experience is similar.

8.4.1 Memoization for datasort inclusion

The first optimization is memoization of subgoals. As observed by Aiken and Murphy [AM91], this is a critical optimization to achieve acceptable performance in implementations of regular tree algorithms.

One complication when implementing memoization is that we have local assumptions, namely the assumptions that "a goal currently in progress is true", as in Chapter 5. Thus, when the algorithm determines the truth of some subgoal, it is in some sense only truth relative to the current assumptions, and it would be unsound to memoize the truth of the subgoal without taking the assumptions into account. This situation is particularly interesting due to the assumptions all being for goals that are currently in progress. We first consider memoization of false results, and then return to the more interesting question of memoization of true results.

Memoization of false results

For subgoals which are determined to be false, the presence of assumptions is not a problem: if a subgoal is determined to be false under some assumptions, then it is false in general, and

we can memoize this result and make use of it regardless of the assumption set. Doing so does alter the results for some subgoals when in contexts containing assumptions that ultimately turn out to be false, but it only modifies true results to false in such contexts, which has no effect on any top level goal.

For example, suppose we have a signature containing the following declarations.

$$\begin{aligned} a_{nat} &= c_z\, 1 \sqcup c_s\, a_{nat} \\ r_{even} &= c_z\, 1 \sqcup c_s\, r_{odd} \\ r_{odd} &= c_s\, r_{even} \end{aligned}$$

If we use the algorithm without memoization to check the goal $r_{even} \trianglelefteq r_{odd}$. Then we make the assumption $r_{even} \trianglelefteq r_{odd}$, and under this assumption we check the subgoals $1 \trianglelefteq \cdot$ and $r_{odd} \trianglelefteq r_{even}$. If we check the second subgoal first, we find that it is true in one more step, using the assumption, but the first subgoal is false, and we determine that the original goal is false.

Now, suppose we modify the algorithm to use memoization of false results. Further suppose that we have previously checked the goal $r_{odd} \trianglelefteq r_{even}$, and memoized the fact that it is false. If we check $r_{even} \trianglelefteq r_{odd}$ the subgoal $r_{odd} \trianglelefteq r_{even}$ matches the memoized result, and the algorithm determines that this subgoal is false, rather than true as in the algorithm without memoization. But, this difference has no effect: it occurs because the assumption $r_{even} \trianglelefteq r_{odd}$ is a "goal in progress" that is ultimately false. When we have such an assumption, we must be in the process of determining that it is false, so there is no harm replacing true results by false results: for that subgoal we will still obtain false. (This argument depends on the fact that our algorithm behaves monotonically, i.e. replacing true by false can only change some true subgoals to false.)

We thus use a hash table to memoize every goal of the form $\rho \leq u$ that is determined to be false using the algorithm in Chapter 5. Goals of this form are convenient memoization points in the implementation, since we have a unique sort name for each base sort, and we can represent ρ as as set of sort names, and u as a set of sets of sort names, and then represent these sets as sorted lists.

We could extend our memoization scheme to all goals $R \leq U$ encountered by the algorithm. We chose not to do this to avoid the cost of calculating hash codes and equality checks for sets of structured sorts at every step. Also, the gain seems small: any goal which "hits" the memo table with such a scheme should also hit under our scheme, except that it will hit a little later when subgoals that involve refinements of datatypes are reached.

In some cases there may be a huge number of such subgoals, such when they are generated via rules for unions of products in the algorithm in Chapter 5. In such cases memoizing all goals might conceivably avoid duplicating a large amount of work, and should such cases prove important in practice then this technique should be considered. The implementation of regular tree inclusion by Hosoya, Vouillon and Pierce [HVP00] does memoize all subgoals, and they observe that hash consing can be used to reduce the overhead of looking up structured sorts at every step.

Memoization of true results

Now, so far we have only considered memoization of false subgoals. To memoize subgoals with true results we need to consider the assumptions present during evaluation of the subgoals. One possibility is to only memoize when there are no assumptions, but this seems to greatly limit the number of true results that are memoized.

One alternative is to take advantage of the fact that all false subgoals are memoized to allow memoization of true subgoals. This is done by evaluating a top-level goal as usual, and then if the goal is found to be true, it is repeated but with a flag set to indicate that true subgoals should be memoized. Since every false subgoal is memoized in the first evaluation, the only paths followed are those for true goals, and so all of the assumptions made are actually true (since assumptions correspond to "goals in progress"). Thus, these assumptions can be discharged, and every true result during the second evaluation is valid without any assumptions, and can thus be memoized. It is convenient to memoize these in the same table as false results, so that we only need to lookup in one table.

In the worst case, the repetition of evaluation for true results takes the same time as the original call, but the memoized results may significantly reduce the amount of duplicated work done later by the algorithm. Until recently our implementation followed this approach, and achieved sufficient performance for all examples that we tested, so we did not consider it necessary to consider other approaches.

One example recently motivated us to consider further improvements our implementation, and in particular to attempt to improve the memoization of true results in our implementation. This occurred during the purely functional lists experiment, described in Section 9.4. For this example, our implementation would previously get stuck when solving a particular datasort inclusion goal, not finishing even when left for 12 hours.

To determine whether this was due to a weakness in our implementation, or due to the complexity of the example, we translated the relevant datasort declarations into both the XDuce language of Hosoya, Vouillon and Pierce [HVP00] and the CDuce language of Benzaken, Castagna, and Frisch[BCF03]. We then constructed appropriate expressions in those languages to force checking of inclusions corresponding to the problematic inclusion. We found that both the XDuce and the CDuce implementations were able to quickly determine the validity of the inclusion.

In an attempt to obtain acceptable performance for this example we made a detailed comparison between the memoization of true subgoals with our approach and with the approach used by Hosoya, Vouillon and Pierce [HVP00] in the XDuce implementation. Our conclusion was that both approaches appear to memoize the same results at the end of each top level goal. However, the XDuce approach has the advantage that it allows reuse of results within subgoals. In particular, it stores all true results immediately, and uses local data structures to store these rather than a global memo table to avoid propagating memoized results in the case when an assumption is determined to be false. This has the potential to avoid a large amount of unnecessary duplication of work while solving a particular goal, compared to the approach of delaying memoization until the top level goal has completed.

Based on this analysis, we decide to modify our implementation so that it stores true results in local data structures in the similar way to the XDuce implementation. We differ in that we still store these results in a hash table at the end of each top level goal. Alas, this modification did not have the desired effect: the performance was still not satisfactory for the example mentioned above. We later found a quite different optimization unrelated to memoization that did have the desired effect (see Section 8.4.2), and also uncovered a number of "performance bugs" in our implementation in the process. We have chosen to retain the XDuce approach to memoization of true subgoals anyway, because it seems that there could be some situations where it avoids a large amount much duplication of work, and it seems that does not ever result

in a noticeable cost in terms of time or space. [1]

It appears that even more true results could be memoized by tracking which assumptions are actually used in the evaluation of each subgoal. This would allow memoization of a true subgoal that occurs as part of a goal that is determined to be false, provided the true subgoal does not depend on the assumption for the goal.

Alternatively, we could attempt to modify the algorithm to avoid the use of local assumptions altogether, as is done in the implementation of CDuce by Benzaken, Castagna, and Frisch[BCF03], who use instead a local solver for Boolean constraints. Their approach does not appear to have been described in detail, but it seems that they assign a Boolean variable for the result of each subgoal, and then they look up and share results via these variables, including when a goal is encountered that is currently in progress. In some sense this approach uses variables to allow sharing of results before they have even been calculated.

Should we encounter further examples for which the current implementation performs poorly, these seem like reasonable directions to pursue.

8.4.2 Optimizations for unions of record sorts

We found that we required some additional optimizations to obtain acceptable performance. In particular we found it necessary to consider optimizations that focus on the interaction between products and unions, since these are mostly responsible for the combinatorial explosion in the number of subgoals. This experience corresponds with that reported in the work of Aiken and Murphy [AM91], the work of Hosoya, Vouillon and Pierce [HVP00] and the work of Benzaken, Castagna, and Frisch[BCF03].

We apply our optimizations to prior to expanding using the rule from Chapter 5, in an attempt to avoid the need for this rule, or at least to reduce the number of subgoals that result. Recall the form of this rule, and the rules for the auxiliary judgment for products.

$$\frac{R \stackrel{\rightarrow}{\simeq} R_1 \otimes R_2 \qquad \Theta \vdash (R_1 \backslash \cdot) \otimes (R_2 \backslash \cdot) \trianglelefteq U}{\Theta \vdash R \trianglelefteq U}$$

$$\frac{S \stackrel{\rightarrow}{\simeq} S_1 \otimes S_2 \quad \Theta \vdash R_1 \backslash (U_1 \sqcup S_1) \otimes R_2 \backslash U_2 \trianglelefteq U \quad \Theta \vdash R_1 \backslash U_1 \otimes R_2 \backslash (U_2 \sqcup S_2) \trianglelefteq U}{\Theta \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq U \sqcup S}$$

$$\frac{\Theta \vdash R_1 \trianglelefteq U_1}{\Theta \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq \cdot} \qquad \frac{\Theta \vdash R_2 \trianglelefteq U_2}{\Theta \vdash R_1 \backslash U_1 \otimes R_2 \backslash U_2 \trianglelefteq \cdot}$$

The code for the implementation of subsorting for records is surprisingly close to these rules (in fact, the rules are originally inspired by the implementation). Each of the optimizations for unions of products applies to a goal of the form $R \leq U$ with R a product sort and U a union of product sorts that is compatible with R.

The first optimization is to check the emptiness of each component of R, and succeed if any component is empty, since then R is empty. The second optimization is to similarly check

[1] In fact, the number of true results stored locally never reached more than 12 while sort checking our standard set of examples. We did manage to construct one contrived example where the number reached over 800.

emptiness of each part of the union U by checking the emptiness of each component, and then remove those parts that are found to be empty because they have an empty component. We found the first optimization to be surprisingly critical in obtaining acceptable performance: without it some of our examples do not terminate after running for many hours. The second has a more minor effect, but consistently improves performance.

The third optimization is to check whether all of R is included in one of the parts of the union U. We found this optimization to also be critical in obtaining acceptable performance.

The fourth optimization is to check whether a component of R is syntactically equal to the corresponding component of all parts of the union U. If so, we ignore this component when determining inclusion. The correctness of this optimization depends upon the fact that we have already checked each component for emptiness. In some cases this optimization can have a major effect, and is particularly important when R and U refine a record type that has a component that has only the default refinement.

The fifth optimization is to check whether either $R_1 \backslash U_1$ or $R_2 \backslash U_2$ is empty before applying the rule for products with a non-empty union. Emptiness of each these differences is equivalent to the subsortings $R_1 \leq U_1$ and $R_2 \leq U_2$ respectively. This optimization has a reasonably large effect on performance: with all other optimizations in place it reduces the total time taken from 36 seconds to 6.7 seconds when analyzing the most complicated datasort declarations that we have encountered so far (see Section 9.4).

We also include one optimization that is based on the implementation by Benzaken, Castagna, and Frisch[BCF03]. This optimization avoids the exponential increase in the number of subgoals for unions of records when one of the fields has disjoint components in each part of the union. For the examples we have tried so far, this optimization has only a small effect, but it seems possible that would avoid a large number of subgoals in some cases.

8.4.3 Other optimizations for datasort inclusion

Another critical optimization in our implementation of datasort inclusion involves the check that determines whether a goal is in the current assumptions. This check is used to determine whether a goal matches one that is currently in progress, and prevents the algorithm from looping when we have recursive datasorts. In Chapter 5 this is the check that determines which of the following two rules is used when we have a goal involving base refinements.

$$\frac{\lfloor \rho \rfloor \leq \lfloor u \rfloor \text{ in } \Theta}{\Theta \vdash \rho \trianglelefteq u}$$

$$\frac{\lfloor \rho \rfloor \leq \lfloor u \rfloor \text{ not in } \Theta \qquad \Theta, \lfloor \rho \rfloor \leq \lfloor u \rfloor \vdash_\Sigma \mathsf{body}_\Sigma(\rho) \trianglelefteq \mathsf{body}_\Sigma(u)}{\Theta \vdash_\Sigma \rho \trianglelefteq u}$$

Our optimization of this check is to match the set of base sorts in $\lfloor \rho \rfloor$ using subset inclusion rather than set equality. Similarly, we represent each $\lfloor u \rfloor$ as a set of sets of base sorts, and match using the appropriate subset inclusions. Thus, we allow the goal to match against an assumption that is strictly stronger. To compare the current goal $\lfloor \rho_g \rfloor \trianglelefteq \lfloor u_g \rfloor$ against an assumption $\lfloor \rho_a \rfloor \trianglelefteq \lfloor u_a \rfloor$ we check whether:

$$\mathsf{sortnames}(\rho_a) \subseteq \mathsf{sortnames}(\rho_g)$$

and:

$$\forall \rho_1 \text{ in } u_a . \exists \rho_2 \text{ in } u_g . \mathsf{sortnames}(\rho_2) \subseteq \mathsf{sortnames}(\rho_1)$$

To efficiently check subset inclusion between the sets of sort names in each base refinement ρ, we represent these sets as sorted lists, using the unique integer identifier attached to each sort name to determine the order. (We also use these sorted sets for the memoization of datasort inclusion results, although in that case we match using equality rather than subset due to the potentially huge number of entries in the memoization table.)

The optimization of matching via subset inclusion potentially has a dramatic effect on performance for recursive datasorts because it reduces the number of "unrollings" via $\mathsf{body}_\Sigma(\rho)$ that are required before matching against an assumption. We found that this optimization reduced the total time taken to analyze datasort declarations by a factor of about two for each of our more complicated examples. For one of our more contrived examples (natural numbers with refinements that each exclude a particular number) the factor is around eight.

We also tested an extension of this optimization which also used subsort inclusion on sort names when comparing elements (although some inclusions will not yet have been determined if we are in the process of analyzing datasort declarations). Alas, this actually made the analysis slightly slower for our test examples, because it requires us to test each element in one set against every element in the other. With the simple subset test, we can instead take advantage of the fact that the list is sorted to implement a faster subset test that takes time linear in the sizes of the lists. Occasionally these sets are large enough for this difference to be noticeable, and since this optimization does not seem to catch many cases, we have chosen not to include it.

8.4.4 Minimizing datasort inclusion tests

The final set of optimizations are not optimizations to the datasort inclusion algorithm itself, but rather avoiding the need to call it in some cases. In particular, during the lattice creation as described for each new datasort we need to construct each intersection with an existing lattice element, and then call the datasort inclusion algorithm to compare each such intersection with every lattice element. Thus, the work done for each new datasort grows with the square of the size of the lattice.

In many cases it is possible to avoid a large part of this work. When are constructing the intersection of a new datasort with a lattice element that was itself created for an intersection, we attempt to use associativity to calculate the lattice element directly. Clearly this will not always work: intersections with some lattice elements are missing, because have not yet reached them, or because we will add them in the phase of adding intersections of intersections via associativity.

We also avoid calling the datasort inclusion algorithm when comparing an intersection ρ_{new} & ρ_{old} with an element ρ by attempting to use the lattice to determine $\rho \leq \rho_{\text{new}}$ and $\rho \leq \rho_{\text{old}}$. If either of these can be determined to be false, then the result of the comparison is false: ρ_{new} & ρ_{old} can not be equal to ρ. If both can be determined to be true, then we can avoid checking the inclusion in one direction: we know that $\rho \leq \rho_{\text{new}}$ & ρ_{old} so we only need check whether ρ_{new} & $\rho_{\text{old}} \leq \rho$. This optimization approximately halved the time taken for the lattice creation in our experiments, and used together with the previous optimization it results in nearly a three times improvement in performance.

If we wanted to further minimize the use of the datasort inclusion algorithm we could allow the programmer to declare those inclusions between datasorts that they intend to hold, and to only check these inclusions hold. Other inclusions would be assumed to not hold, except when they follow relatively directly from the declarations. It seems likely that in many cases the number of declarations required would be small enough to be practical, and in some cases very few inclusions may be required, or even none. Additionally, these declarations would also document which inclusions are required when sort checking the expressions in the program, which would be useful when considering modifications to datatype and datasort declarations.

We could even go one step further and avoid the use of the datasort inclusion algorithm altogether. To do this we would allow the programmer to declare a set of inclusions between unions of datasorts that is "consistent" in the sense that each declared inclusion can be checked only making use of the declared inclusions for base refinements. This means that we can check each declared inclusion by comparing the datasort bodies using a variant of the datasort inclusion algorithm that does not accumulate assumptions, but instead uses the declarations as the set of inclusions that hold between base refinements. In some sense this is allowing the programmer to declare a "fixed-point" in terms of the inclusions that hold, while the algorithm in Chapter 5 calculates a greatest fixed-point that contains as many inclusions as possible. Checking the declared inclusions would still take exponential time in the worst case, but the exponent is likely to be small, and would be much easier to predict: it would be the maximum number of times that a value constructor is repeated in a single datasort declaration, where that constructor takes a product as an argument.

Another alternative would be to avoid the explicit construction of the lattice of unique refinements for each datatype and instead directly call the implementation of datasort inclusion during the sort checking of expressions. In some sense this would "lazily" construct the lattice: if a particular inclusion is never checked, it is never calculated, and if it is checked then the memoization in the implementation of datasort inclusion will avoid repeating the work done. This scheme does seem attractive, particularly because the implementation of the lattice creation has required some tuning to obtain acceptable performance. However, this scheme would have the disadvantage that we would need represent intersections of base refinements syntactically, instead of having a sort name for each equivalence class of base refinements. This would require some relatively fundamental changes to the implementation. It might reduce the effect of memoization in some cases, since there may be many base refinements which are equivalent but which are formed by intersecting different base sorts.

These seem like reasonable directions to pursue if further experience with the implementation indicates that there are situations where the implementation of datasort inclusion has unsatisfactory performance.

8.5 Subsort checking

The implementation of subsort checking that is used during the sort checking of expressions shares most of its code with the datasort inclusion algorithm. However, when a goal involving base refinements is encountered, the relevant lattice is used to solve the goal directly rather than using assumptions and comparing datasort bodies.

This works because the subsorting goals that are generated while checking expressions do not involve unions, and for such goals our algorithm never generates a subgoal involving unions

(our lattices include intersections, but not unions). Subgoals involving emptiness do arise though, in particular for records, and so we record the emptiness of each base sort as part of the datasort analysis. (Only the minimal element of a lattice can be empty, but it need not be. Also, currently there is no way to declare that a refinement of an opaque type is empty.)

Subsorting is thus implemented as a generalized function that is parameterized by a base subsort function `subSortName` that determines inclusions $\rho \leq u$ involving base refinements. Thus, we use the following type for the base subsort function.

```
subSortName : SortName list * SortName list list -> bool
```

The generalized function is instantiated to the one used during checking of expressions by applying it to a base subsort function that consults the appropriate lattice, or checks emptiness if the list representing the union on the right-hand side is empty. This base subsort function raises an exception if the list representing the union contains more than one element.

Instantiating the generalized subsorting function to the one used during datasort analysis is somewhat more complicated. The function in the datasort analysis that determines inclusion between datasorts calls the generalized subsorting function for each subsorting goal, and passes itself as the function used to solve base subsorting goals. More precisely, the datasort inclusion function is parameterized by a list of assumptions, so it actually instantiates itself by adding an additional assumption for the current goal, following the algorithm in Chapter 5, and then passes the resulting function to the generalized base subsorting function.

8.6 Backtracking and error messages

The implementation of sort checking for expressions is based on the bidirectional algorithm in Chapter 3, which involves backtracking. This is deep backtracking, meaning that after a goal has returned a successful result, a subsequent failure may require revisiting the goal to generate further results. Since SML expressions can contain core-level declarations, this backtracking extends to core-level declarations also. This issue is complicated by the fact that the sort checker should produce error messages upon failure, and should generally attempt to continue in order to produce more than one error message.

Our approach is to implement a small library that contains a type constructor for computations that may fail and produce a list of errors, and a type constructor for computations that may backtrack. Composing these two type constructors yields a type constructor for computations with backtracking and errors. We compose this type constructor from the two simpler ones so that we can use the just type constructor for errors in when backtracking is not appropriate. For example, only the judgment for expressions that synthesizes sorts requires backtracking, but the judgment that checks an expression against a sort does not. Similarly, the analysis of datasort declarations may return errors, but does not backtrack.

These two type constructors are implemented in a module `Comp` in our implementation, which also includes a number of combinators for building backtracking and non-backtracking computations.

8.6.1 Computations with errors

The signature for the module `Comp` contains the following specifications for the type constructor for computations with errors.

```
type Error                              (* Errors to be reported. *)
exception Fail             (* Raised when not reporting errors. *)

type 'a Result = 'a * Error list    (* Results of computations. *)
type retErrs = bool                (* Whether to return errors. *)

(* A type constructor for computations which may fail.  When the
   bool argument is true, failure is indicated by a non-empty
   list of errors, otherwise it is indicated by raising Fail.
*)
type 'a Comp = retErrs -> 'a Result
```

Exposing that the type constructor for computations is implemented as a function is convenient: functions that construct computations can be implemented by adding an additional curried argument of type `bool` which is passed on to any subcomputations as appropriate. We could have instead kept the type constructor for computations abstract, and instead provided a constructor like the following.

```
makeComp : (retErrs -> 'a Result) -> 'a Comp
```

This would have been somewhat more notationally cumbersome, and it is not clear that much would be gained.

The type of results is simply the underlying type paired with a list of errors. Each error corresponds to one error message that is reported. When a result is returned with an empty list of errors, the computation has succeeded. The reason we include a value of the underlying type even when the computation fails is that we would like the sort checker to continue checking after it finds an error, so that more than one error can be reported. This means that failing computations need to provide some reasonable return value when they fail. The representation of sorts includes a "bogus" constructor for exactly this purpose: it is considered to be compatible with every other sort, and is only used by the sort checker when a sort error has been found.

The reason that we require computations to raise exceptions in some cases and return a list of errors in other cases is that in many cases we are not interested in the errors, and in such cases it would be inefficient for a failing computation to continue. For example, suppose we are checking an expression against a sort constraint with alternatives, such as the following.

```
(*[ val twoIsFour <: tt, ff ]*)
val twoIsFour = (2 = 4)
```

For this example, our implementation would first check the expression against the first alternative, and if that fails (or we later backtrack) it would then try the second alternative. However, to avoid returning multiple errors for the same program location, we only return the errors for the second alternative. Thus, there is no point in accumulating errors while trying the first alternative, so that computation is started by passing false. Then, if at any point the first computation fails it raises the exception `Fail`, which we can catch and immediately

proceed to trying the second alternative. Rather than implement this pattern each time we have multiple alternatives to try, the library includes two combinators `tryBothC1` and `tryBothC2` which combine two computations into one that first tries the first computation and if that fails the second is tried. If both fail `tryBothC1` returns the errors of the first computation while `tryBothC2` returns the errors of the second.

8.6.2 Backtracking computations

Backtracking computations are represented using the type constructor `RComp`, which is specified as follows in the signature for the module `Comp`.

```
(* Results that can be "backtracked" by via a computation that
   calculates the next result.  Do not request error reporting
   when initiating the computation of the next result.
*)
datatype 'a Redo = REDO of 'a * ('a Redo) Comp

(* Computations with backtracking. *)
type 'a RComp = 'a Redo Comp
```

The type constructor `Redo` could be made opaque, since no part of the sort checking code depends upon it being transparent. However, we feel that exposing this datatype declaration is likely to assist in understanding the purpose of these two type constructors and the various combinators in the library related to backtracking.

The type constructor `RComp` is specified as computations which have a result that is "redo-able". A redo-able result is a pair containing the result and a computation that can be used to calculate the next result. Thus, `RComp` is essentially a type constructor for lists of results, with the whole list and each tail being delayed via the `Comp` type constructor, and thus incorporating failure with errors. When one of the "redo" computations is evaluated to calculate a further result after a success, no list of errors should be requested, and the lack of further results will always be indicated by raising the exception `Fail`. This is appropriate because there are only errors that should be reported when the whole computation does not return even one success, and in this case the list of errors for the original computation can be used.

8.6.3 Combinators

The library of computations contains various combinators for constructing and composing computations and backtracking computations. Some of the most important combinators are combinators that embed ordinary values into the types for successful and non-backtracking computations, and higher-order "let" forms `letC` and `letR` that sequence two computations, passing each result of the first computation to a function that constructs the second computation. Each of our type constructors could be considered to have the structure of a monad with respect to these combinators (see the work of Moggi [Mog91]). However, these combinators have no particularly special status in our implementation, and we originally included them because they seemed natural, and only later realized the relationship to monads.

Other combinators include `tryBothR1` and `tryBothR2` which combine alternative backtracking computations in a similar way to `tryBothC1` and `tryBothC2`, but which include backtracking. We also have combinators that take a list of errors and fail immediately, one that generates

a list of all successful results for a backtracking computation, combinators that adds memoization to backtracking computation and non-backtracking computations, and a number of similar combinators that were found to be convenient in the implementation of the sort checker.

One interesting combinator is a variant of a let called `letRC` that sequences a backtracking computation with a non-backtracking one, and constructs a non-backtracking computation, but one that may backtrack over the first computation if the second one fails. This combinator made the interface between backtracking and non-backtracking code particularly simple. This interface is also simplified by the fact that the type constructor for backtracking computations is an instance of the type constructor for non-backtracking computations.

An alternative approach to simplifying this interface would be to use a single type constructor for both backtracking and non-backtracking computations, essentially by adding a non-backtracking version of `tryBothR`. We did not follow this approach because non-backtracking computations are used more often than backtracking ones in our implementation, and so we felt that treating them as a special case of backtracking computations was inappropriate. Also, it seems that something like `letRC` would still be required with such an approach, to stop subcomputations of non-backtracking computations from backtracking unnecessarily, and it would have been more difficult to recognize this if we had only a single type constructor for computations.

8.6.4 Comparison with other approaches to backtracking

Our approach to implementing backtracking has some similarities to that used by Paulson in the implementation of Isabelle [Pau86], particularly in that both use a form of list with each tail being a delayed computation. We experimented with a number of alternative approaches and variations before choosing this approach. The simplest approach seems to be to represent backtracking by returning a standard list of results. This approach has the major disadvantage that all results must be calculated, when often only the first is required, so generally leads to much worse performance.

The most notable alternative that we considered was to pass to each computation a continuation function to call with each successful result, and for computations to return normally when there are no more results. This is the approach is used by Carlsson [Car84] to to implement deep backtracking in an implementation of logic programming via a functional language, a technique that was also used by Elliot and Pfenning [EP91]. This approach to backtracking is somewhat dual to ours: we have "redo" continuations for the case where a computation succeeds, but a subsequent computation fails and requires a the goal to be revisited. In the case of implementing logic programming, success continuations appear to be particularly appropriate, but in our case they seem less so. Mostly this is because using success continuations leads to a slightly less direct style of programming. This is particularly noticeable for computations that require failure with errors, but do not require backtracking, which is the more common case in our implementation.

We also considered implementing our backtracking and non-backtracking computations as monads using the approach proposed by Filinski [Fil94, Fil99]. This approach would have the advantage that the ordinary SML let could be used to sequence computations, allowing a slightly more direct style of programming, and likely also reducing the performance overhead of sequencing computations. We decided not to use this approach because it requires an extension of SML with a feature that captures the current continuation as a function, and we did not

want our implementation to depend on such non-standard features. Also, our "let" combinators appear to allow a sufficiently natural style of programming, and the overhead involved does not appear to be a dominant factor in our sort checker.

8.7 Sort checking for core-level declarations and expressions

The implementation of sort checking for core-level declarations largely follows the corresponding code in the elaboration phase in the ML Kit. The part that deals with expressions is based on the bidirectional checking algorithm described in Chapter 3 and extended with pattern matching in Chapter 6. Backtracking and failure in this algorithm are implemented using the computations with backtracking and errors described in Section 8.6. Thus, our mechanism for reporting errors does not follow the ML Kit elaborator, and as a result our code does not have to rebuild an abstract syntax tree, so is somewhat simpler.

8.7.1 Expressions

The main part of the code for sort checking expressions is based on the following functions:

```
val check_exp : Context -> exp -> unit Comp
val infer_exp : Context -> exp -> Sort RComp
```

We also have the corresponding functions for atomic expressions, since these are represented using a different type in the ML Kit. The code for checking and inferring sorts for expressions is quite directly based on the rules in 3, with the following notable differences.

- Upon failure we generate an error value containing appropriate information.
- We allow an application of a function to an inferable expression as a checkable form. This required because case expressions in SML are translated to such expressions.
- When we encounter an non-inferable expression while inferring, we check the expression against the default refinement, as described in 7.
- We also have records, pattern matching, parametric polymorphism, exceptions, special constants and let expressions introducing core-level declarations. The extension to these features is mostly straightforward, following Chapter 6, Chapter 7.5 and Chapter 7. We describe the implementation of checking for pattern matching and core-level declarations in detail below.

8.7.2 Pattern matching

In SML, functions may involve pattern matching via a sequence of branches, each with an argument pattern and a result expression. The implementation of sort checking for such pattern matching is closely based on the algorithm in Chapter 6. Recall that this algorithm requires the notion of pattern sorts, which are a generalization of sorts to include applications of constructors and a form of union.

$$\text{Pattern Sort} \quad Z \quad ::= \quad R \mid c\,Z \mid (Z_1, Z_2) \mid () \mid Z_1 \sqcup Z_2 \mid \bot$$

The implementation of sort checking for pattern matching is based on the following judgments from Chapter 6.

$\Pi \vdash_\Sigma Z \bowtie \Omega^{\mathsf{c}} \overset{\Leftarrow}{\in} R$	Matching pattern sort Z with branches Ω^{c} checks against sort R under context Π.
$\Pi; \Delta \vdash_\Sigma C \overset{\Leftarrow}{\in} R$	Term C checks against sort R under context Π and pattern context Δ.
$\vdash_\Sigma Z \overset{\Rightarrow}{\preceq} Y$	Y is a minimum sort union covering pattern sort Z.

The major differences from the formal presentation in Chapter 6 are the following.

- In the formal presentation, matching a pattern against a pattern sort is done via the following goal.

  $$\Pi; P \in Z \vdash C \overset{\Leftarrow}{\in} R$$

  Via the rules for this judgment, this ultimately involves checking the expression C against R under some finite set of contexts that extend Π, i.e. we check

  $$\Pi, \Pi_i \vdash C \overset{\Leftarrow}{\in} R$$

  for some number of standard (non-pattern) contexts Π_i.

  In the implementation, we instead have a function `ref_pat` which takes P and Z as input and returns a list of contexts under which C needs to be checked. The advantage is that we can remove redundant contexts from this list, and thus avoid unnecessarily checking C against some contexts. Also, this function avoids the need for a special kind of context for assumptions about the sorts of patterns.

- The function `ref_pat` additionally returns a corresponding pattern sort Z_i for each context Π_i in the result. This pattern sort is the least one such that the assumptions in Π_i imply that the pattern P is contained in Z_i. These pattern sorts are used to assign sorts for layered patterns, and also to check against sort constraints in patterns, as specified in Chapter 7.

- This function also calculates pattern sorts for the difference $Z \backslash P$ and pattern intersection $Z \wedge P$. The first is needed either as the input pattern sort for the subsequent branch, or for the emptiness check after the last branch. The second is required for in order to calculate the first in the case of products, if we use the alternative definition of $Z \backslash P$ for products in Section 6.5. Calculating these in `ref_pat` avoids some duplication of work that would occur if we implemented them as separate operations.

- We simplify pattern sorts containing occurrences of the empty pattern sort $\bot$ and empty sorts "on the fly" as they are constructed. As a result empty sorts can never appear in pattern sorts, and $\bot$ is eliminated completely except for top level occurrences.

  This optimization is critical. With the optimization, the largest pattern sort we have encountered so far has 378 nodes (in the operator fixity resolution example). Without it, the size of pattern sorts is so large as to make sort checking infeasible for some of our examples: for these the size quickly grows to the many millions, and progress quickly comes to a halt once all available RAM has been consumed.

Putting together the first three points, the main function for in the implementation of pattern matching has the following type.

```
val ref_pat : Context -> pat -> PatSort ->
               ((VarEnv * PatSort) list * PatSort * PatSort) Comp
```

There is a corresponding, mutually recursive function for atomic patterns.

8.7.3 Value declarations

Sort checking for value declarations is done by the following function.

```
val ref_valbind : Context * valbind -> VarEnv RComp
```

This function is a backtracking computation, since there may be many sorts inferred for the expressions in the declaration.

This function involves somewhat more than just calling `infer_exp` to infer the sort of the expression. It first checks whether there is a sort constraint at the top of each pattern, or if there is value sort specification for the variable bound in the pattern (as described in Section 7.1). If so, it backtracks over the comma-separated alternatives, and checks the expression against each.

It then matches the pattern against each sort obtained for the expression, to obtain a variable environment containing the sorts for each variable in the pattern. If there are unmatched cases, we fail with a warning.

In some rare cases for value declarations that include patterns with constructors, there may be a case analysis required when matching the pattern with a sort, in which case an error is reported, since in general there is no single variable environment that would correspond to the required case analysis (roughly what is needed is a union). Such value declarations cases can always be transformed into equivalent declarations with expressions that use `case`, resulting in the required case analysis being performed over the body of the case. For example, consider the following declarations.

```
datatype d = C1 | C2 of bool
(*[ datasort d2 = C2 of tt | C2 of ff ]*)

(*[ val f <: unit -> d2 ]*)
fun f () = C2 true

val (C2 x) = f ()
```

The final declaration requires a case analysis: the expression has sort `d2` and so we need to consider both the case where `x <: tt` and where `x <: ff`. Backtracking is not what we want here: backtracking determines whether one of alternatives results in a success, while here we need to analyze the whole scope of `x` and check that each part of the case analysis would result in success. In the absence of some form of union, it seems that the only way to handle this is to consider it an error. The problematic declaration can be transformed into the following one using `case`.

```
(*[ val x <: bool ]*)
val x = case f () of x2 => x2
```

In general, the sort declaration is required, although in this case it is optional since `bool` is the default sort.

8.7.4 Other declarations

Since value declarations require backtracking, the function which performs sort checking of declarations also constructs a backtracking computation. However, the only other declarations that can involve backtracking are `local` and sequential declarations, and these only propagate the backtracking introduced by value declarations.

Otherwise, the sort checking for other SML core-level declarations is quite closely based on the elaborator of the ML Kit.

8.7.5 Memoization

The implementation memoizes results using references attached to the abstract syntax tree. For declarations that do not depend on the sorts assigned to variables, this memoization simply stores the result once it has been calculated, and should we ever check the declaration again, we reuse the stored result. This is particularly important for the costly analysis of datasort declarations.

For some time we were unsure as to whether memoization of results for value declarations and expressions was required to obtain acceptable performance. Such memoization is more difficult, because we need to take account of the assumptions in the value environment, which can be large, and are generally very similar. Even checking equality between a single pair of value environments may take longer than that saved by reusing a memoized result. Since we had not encountered examples where such memoization would be of obvious benefit, we chose not to tackle the problem of how to implement such memoization efficiently.

Motivating example

The following small example convinced us that there are situations that arise in practice where such memoization avoids needlessly repeating huge amounts of work.

```
(*[ datasort 'a nil = nil
         and 'a one = :: of 'a * 'a nil
         and 'a two = :: of 'a * 'a one ]*)

 val ll = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

An expression like this was encountered while running the sort checking on its own source code. The actual expression is in the Timing module of the ML Kit, and contains a list of strings to be concatenated for formatted output. The refinements for lists were actually irrelevant to this expression, but regardless they had a huge effect on the time taken for sort checking: the time taken for this expression is slightly less than an hour (and it might be considered lucky that the list did not contain 20 elements).

To see why this example takes so long to sort check, remember that in SML this expression is syntactic sugar for the following.

```
1::(2::(3::(4::(5::(6::(7::(8::(9::(10::(11::(12::(13::(14::nil))
                                                          ))))))))))
```

Our sort checking algorithm backtracks excessively over this expression, because with the above sort declarations, the sort for the constructor `::` is as follows (this is exactly as reported by the sort checker).

```
con :: <:  'a * 'a one -> 'a two
         & 'a * 'a list -> 'a list
         & 'a * 'a nil -> 'a one
         & 'a * 'a <empty> -> 'a <empty>
```

Thus, to infer the sort for the top level application of `::`, we try each of the four parts of the intersection sort for `::`, and check the arguments of `::` against the corresponding sorts. In each case we check the second argument by inferring the sort and comparing with the goal. Thus, for each goal we try four cases. As we continue down the list, the number of times we check each subexpression grows exponentially, up to 4^{14}, which is roughly 270 million.

Once we added memoization to the checking for expressions, the time taken for this example reduced to less than a tenth of a second (even if we were unlucky and there were with 20 in the list, or even 100). Memoization is not the only way to obtain acceptable performance for this particular example. For example, we might instead require that the sort of the whole expression be declared, and check in a top-down fashion without ever inferring sorts. But, requiring all applications of constructors to appear in checking positions seems limiting, and unnecessary. And such an approach would not apply to case where we had a repeated function application in place of `::`. In general, it seems that memoization is the most natural approach to avoiding repetition of work in situations like this.

Approach

The main challenge in implementing this form of memoization efficiently is to avoid comparing whole value environments for equality each time we check to see whether one of the previous results had a value environment matching the current one. We tested a naïve implementation, and found that it increased the time taken for sort checking for most of our larger examples by more than an order of magnitude.

Our current approach is to restrict attention to those variables declared in the current core-level declaration. This works because we never backtrack beyond a core-level declaration, and the case analysis related to pattern matching only involves a particular expression, hence cannot involve more than a single core-level declaration. We store a memoization table in each node of the abstract syntax tree.

For expressions that we infer sorts for, this table has the value environment as the key, and each value is a memoized version of the backtracking computation used to infer the sorts for the expression. This memoized computation will immediately return a result if at some point it has been calculated, thus is essentially a lazy list (i.e. memoizing stream) with respect to our type constructor computations with errors. For expressions that we only check against sorts,

we use a table that has both the value environment and the sort as the key, and has a memoized version of the corresponding non-backtracking computation as the value. Value declarations are memoized in a similar fashion to inferable expressions.

With some tuning, this approach seems to work well: the overhead for our larger examples is around 10%. However, not much benefit seems to have been gained by memoization for these examples: only very few goals are matched against memoized results. But, the earlier example involving a list clearly shows that this is a critical optimization if we want to obtain acceptable performance for the very wide range of expressions that may be encountered in practice.

Additional opportunities

We could attempt to go much further with memoization. We could restrict attention further to only those variables that actually appear in an expression or declaration. This would decrease the overhead, and also increase the number of matches against previous results, since irrelevant differences between environments would be ignored.

We could also match against previous results using subsorting for both goal sorts and the sorts in the environments, instead of equality as we do currently. If we did this, it seems that memoization might remove the need for some of our other optimizations, for example the removal of redundant contexts during pattern matching. It also seems that some more experiments are warranted: for our current set of examples there seem to be too few goals in total for memoization to have much benefit, and the overhead might conceivably prove to be more of an issue with larger numbers of memoized results in each table.

We plan to investigate these issues further in future work, based on further experience with the current implementation.

8.8 Sort checking for the SML module system

Rather than implement a separate phase for sort checking after all elaboration for modules has completed, we decided to instead to integrate sort checking directly into the module-level elaboration code of the ML Kit. This avoids a lot of duplication of code, particularly since there is relatively little code that is specific to refinements at the module level (unlike the core level). Also, an integrated approach is somewhat easier at the module level than the core level because we do not need to worry about the elaborator using effects to implement unification.

Our modifications to the module-level elaboration code also required extending the representations of module-level environments used by that code. In particular, each occurrence of a core-level environment in these representations was replaced by a pair containing the original environment and a corresponding environment for sorts. The environment for sorts should *refine* the original environment, meaning that it should have the same structure, but contain sorts that refine the types in the environment for types. [2] Similarly, each occurrence of a type

[2] Tracking bugs that result in violations of this property proved somewhat difficult. An essential technique was to use debugging "assertions" that check this property each time an environment is constructed, but only when running in a "debugging mode" since these tests are time consuming. Even with this technique, finding such bugs was tedious, particularly when tracking a violation of the property observed in a large test case. Additionally, there may be bugs that we are not aware of because they have not yet been triggered by the examples we have tried. Statically checking this invariant would have been very useful, and should be possible using some form of dependent refinements, as in the work of Xi and Pfenning [XP99] and Dunfield and Pfenning [DP04].

name set in the original representations was replaced by a pair containing the original set and a corresponding type name environment that contains the required information about the refinements of each type name in the set. These type name sets are used by the ML Kit to tracking the binding of type names, following the style of the Definition of Standard ML [MTHM97].

The core-level sort checking for a declaration is performed immediately after the core-level elaboration for the same phrase has completed. Any errors during core-level sort checking are returned via our computations with errors, as described in Section 8.6. These errors are attached to top node of the abstract syntax tree for the declaration, in order to integrate with the ML Kit error reporting mechanism. We added a special kind of "elaboration error" to the ML Kit to support this integration: such an error contains a non-empty list of sort checking errors for a declaration. Each of these sort checking error carries its own information about the location in the source code where the error was detected, unlike ordinary ML Kit errors which rely on the position in the AST where the error is attached. Warnings during sort checking are separated out from errors at this point and reported via the ML Kit warning mechanism.

To avoid violating the invariants upon which the sort checking code relies, we avoid invoking the core-level sort checking code if an elaboration error has previously occurred. We do this by checking a reference that is set whenever an elaboration error is generated. We similarly check this reference prior to invoking module-level sort checking code that depends upon invariants involving the absence of elaboration errors. This is somewhat more awkward than in the core-level sort checking code, but overall works quite well.

Other interesting features of this part of the implementation include the following.

- Where we have a type realization in the original code (a map from type names to type functions, as in the Definition of SML), we add a corresponding sort realization. The sort realization maps each sort name that refines a type name to a sort function that refines the corresponding type function.

- The implementation of these sort realizations is closely based on ML Kit implementation of type realizations. One difference is that the sort names which are created in order to represent intersections are generally omitted from sort realizations. Instead, when applying a sort realization to such a sort name we find the sort names that were intersected to create the sort name, and form the intersection of the realizations for those sortnames. When applying a realization to a sort, this may result in a sort that is not in simplified form, so we also simplify the resulting sort.

  In general, a sort name s may be formed as an intersection in many different ways, for example we may have $s = s_1 \,\&\, s_2$ and $s = r_1 \,\&\, r_2$. To ensure that all possibilities lead to equivalent results, sort realizations must always be preserve the lattice structure of the refinements of each type name. More precisely, a sort realization ρ must satisfy the following sort equivalence for all compatible sort names r and s.

  $$\rho(r \,\&\, s) = \rho(r) \,\&\, \rho(s)$$

  For our example, this means that the intersection of the realizations $\rho(s_1)$ and $\rho(s_2)$ must be equivalent to the intersection of the realizations $\rho(r_1)$ and $\rho(r_2)$, since both must be equivalent to $\rho(s)$.

- When matching against the refinements of an opaque type, we need to check that the sorts realizing the refinements satisfy the specified subsort inclusions (they are allowed to satisfy more). We do this by checking that the above equation holds for all refinements r and s of the opaque type.

- To construct the lattice of refinements of an opaque type in a signature, each time a sort specification is encountered we add new lattice elements for the sort and for the intersection of the sort with every other lattice element. This doubles the size of the lattice (although it less that doubles it when the sort specification is of the alternative form that includes an upper bound).

  When a subsort specification is encountered, we generate a realization that "collapses" the lattice according to the equalities between sorts that follow from the specified subsorting. This realization turns out to be relatively easy to calculate: for the subsorting $r < s$ we map every subsort r' of r to $r' \,\&\, s$. Doing so leads to awkward names for sorts that involve excessive intersections, so we then rename these sorts using a left inverse of this realization, which generally maps $r \,\&\, s$ to r. This means that the whole realization maps r to r, but equates sorts so that the structure of the subsorts of r matches the structure of the subsorts of $r \,\&\, s$ in the original lattice.

- When matching signatures, we allow subsorting between the respective sort schemes for a particular value, as described in Chapter 7. We also allow one sort scheme to be more general that the other, since Standard ML allows a more general type scheme.

Aside from these points, the code for sort checking at the level of modules is closely based on the code of the original elaborator.

Chapter 9

Experiments

In this chapter we describe some experiments with programming with the refinements and implementation described in the previous chapters. Our focus is mostly on what can be achieved using refinements, rather than on testing the performance of our implementation, except where the performance affects the usability of the sort checker. However, in order to give an impression of the typical performance of the implementation, the following table summarizes the total number of lines of code for each experiment, the number of lines of sort annotations, the number of datasorts declared, and the time taken for sort checking. All times were measured on a 1.7GHz Pentium M, and are the average of five runs. Each time is less than half a second, indicating that performance is not an issue for the code of any of these experiments (although a performance issue did arise in the last experiment, which will be described in Section 9.4).

	Total lines	Ann. lines	Datasorts	Time (sec.)
Red-black trees	233	29	6	0.12
Normal forms	70	20	4	0.05
Operator resolution	1451	138	18	0.44
Purely functional lists	143	32	12	0.08

Each of these experiments is described in detail in the remainder of this chapter. In order to give a concrete view of the use of refinements in practice, we present a reasonable portion of the actual code with refinement annotations for each experiment. We also briefly comment on the result of each experiment.

The full code for these experiments, including the module-level code, is available from the following web page. Additional examples will be added to this page.

`http://www.cs.cmu.edu/~rowan/sorts.html`

The first three of these experiments were conducted with the assistance of Frank Pfenning, and the fourth was conducted by Kevin Watkins, both of whom I would like to thank.

9.1 Red-black trees

This experiment involved adding sorts to an implementation of red-black balanced binary search trees. This experiment was carried out with the assistance of Frank Pfenning, who is the

author of the code. This code is part of the Twelf implementation, and is also used in the implementation of the sort checker itself.

Red-black trees are guaranteed to be "nearly" balanced via two invariants involving the colors assigned to nodes. Each node is either red or black. The coloring invariant states that a red node may not have a red child. The balancing invariant states that every path from the root to a leaf passes through the same number of black nodes. Together, these invariants imply that the length of the longest path to a leaf is at most twice the length of the shortest, since at least every second node on such a path must be black.

In our experiment we only represented and checked the coloring invariant. The balancing invariant can not be captured using our datasort declarations, since it would require an infinite number of refinements: one for each possible path length.

This experiment is particularly interesting because the result can be compared with other work on statically checking the invariants of red-black trees. This includes the work of Xi [Xi98], who uses dependent refinements indexed by integers to statically check the invariants of red-black trees. That work is based on the same implementation of red-black trees as ours, so is particularly easy to compare. Overall, index refinements seem particularly natural for capturing the balancing invariant, but we feel that our approach more naturally captures the coloring invariant. The combination of index refinements with intersection refinements, as proposed by Dunfield and Pfenning [DP04], would allow a the natural expression of both invariants. Capturing the invariants in a natural way is very important because it makes it easy for programmers to write and read their annotations, and also because it allows for informative feedback when errors are discovered.

Other work on statically checking red-black trees includes that of Kahrs [Kah01] who uses higher-order nested datatypes, phantom types and existential type variables in Haskell to enforce the balancing invariant of red-black trees, and uses separate types for red and black nodes to enforce the coloring invariant. The main disadvantage of this approach to the coloring invariant is that it requires separate versions of each function for each of type. This makes the code longer, more complicated, and slightly less efficient. In our approach, such functions are instead assigned intersection sorts, and we did not need to modify the underlying code at all.

This is also a particularly good example of the need for intersections in refinements: the function that inserts an item in a tree involves a recursion that may temporarily violate the invariant in a particular way, but it will never violate the invariant if the original tree has a black node at the root. When conducting this experiment, it took a little time to reconstruct these invariants: the comment documenting them did not quite match the actual code. Had a sort checking implementation been available as the code was written, this problem could have been caught. We found the feedback generated by the sort checker to be very useful while reconstructing these invariants, an unanticipated successful aspect of this experiment. Overall, this experiment must be considered a success: we were able to elegantly express the desired invariants using sorts and check them using the implementation.

The code for the data type for red-black trees and the refinements required to check the coloring invariant appear in Figure 9.1. The code for inserting an item into a red-black tree appears in Figure 9.2. This includes a local function `ins1` that requires a sort involving an intersection. The first part of the intersection expresses the fact that the result of insertion may violate the red-black invariant, but only at the root. The second part expresses the fact that no such violation will occur if the input tree has a black node at the root. Only the first

is required for the external call to `ins1`, but the second is required in order to obtain the first: it is needed for the recursive calls. This use of a stronger sort than is required externally is somewhat reminiscent to the proof technique of "strengthening" an induction hypothesis. The only modification that have made to the code for `insert` is to add an explicit scope for the type variable `'a`, which is required to correctly determine the scope of the type variable in the annotation for `ins1`.

We also include the code for the function `restore_right` which performs rotations on the tree to restore the coloring invariant. We omit the corresponding function `restore_left`, since it is symmetric.

We have left the original comments in the code, even though in some cases they are made redundant by the sort annotations. A gray background is used for sort annotations in the code for the function definitions to help emphasize which parts are annotations for the sort checker.[1]

Chris Okasaki has recently proposed a simpler formulation of the rotations that restore the coloring invariant [Oka99] that are particular appropriate in a functional setting. This formulation of the rotations is also the one used by Kahrs [Kah01]. As an additional experiment, we tried modifying the red-black tree code to use this simpler formulation. We did so in a relatively naïve fashion: we did not even attempt to check whether the datasort declarations required modification. Instead we adopted an optimistic approach and modified the code for the required functions, and then sort checked them to see whether the previous datasort declarations would suffice. This seems likely to be an approach adopted by many programmers in practice.

The result of this sub-experiment was that the modified code was accepted without the need to revisit the datasort declarations. The modified code for `restore_right` appears in Figure 9.1, (`restore_left` is symmetric).

9.2 Normal forms with explicit substitutions

Our second experiment involves using sorts to capture normal forms and weak head-normal forms in a λ-calculus with explicit substitutions. We mostly let the code for this experiment speak for itself: the desired invariants can be expressed in natural way. We observe that intersections are not required in this example, thus our refinements are useful in practice without intersections, at least in some circumstances.

The datatype and datasort declarations for this experiment appear in Figure 9.4, which also includes the functions for applying substitutions and composing substitutions. The normalization functions appear in Figure 9.4, which also includes some examples with appropriate sorts assigned.

This experiment was carried out by Frank Pfenning. Overall it must be considered a success: the desired invariants were elegantly expressed by sorts, and checked by the sort checker.

9.3 Twelf operator fixity and precedence resolution

This experiment involved using sorts to check some relatively complicated invariants in part of the parser used in the Twelf implementation. This code resolves operator fixity and precedence,

[1] A similar means of distinguishing sort annotations seems useful while programming, and we have built an extension of the emacs SML mode for this purpose, which is included with the implementation.

```
  datatype 'a dict =
    Empty                (* Empty is considered black *)
  | Black of 'a entry * 'a dict * 'a dict
  | Red of 'a entry * 'a dict * 'a dict

(*[
    (* Trees with only black children for red nodes *)
    datasort 'a rbt =
        Empty
      | Black of 'a entry * 'a rbt * 'a rbt
      | Red of 'a entry * 'a bt * 'a bt

    (* As above but additionally the root node is black *)
    and 'a bt =
        Empty
      | Black of 'a entry * 'a rbt * 'a rbt

    (* Trees with a red root node *)
    datasort 'a red =
        Red of 'a entry * 'a bt * 'a bt

    (* invariant possibly violated at the root *)
    datasort 'a badRoot =
        Empty
      | Black of 'a entry * 'a rbt * 'a rbt
      | Red of 'a entry * 'a rbt * 'a bt
      | Red of 'a entry * 'a bt * 'a rbt

    (* invariant possibly violated at the left child *)
    datasort 'a badLeft =
        Empty
      | Black of 'a entry * 'a rbt * 'a rbt
      | Red of 'a entry * 'a bt * 'a bt
      | Black of 'a entry * 'a badRoot * 'a rbt

    (* invariant possibly violated at the right child *)
    datasort 'a badRight =
        Empty
      | Black of 'a entry * 'a rbt * 'a rbt
      | Red of 'a entry * 'a bt * 'a bt
      | Black of 'a entry * 'a rbt * 'a badRoot
]*)
```

Figure 9.1: Datatype and datasort declarations for red-black trees

```
  (* restore_right (Black(e,l,r)) >=> dict
     where (1) Black(e,l,r) is ordered,
    (2) Black(e,l,r) has black height n,
    (3) color invariant may be violated at the root of r:
        one of its children might be red.
      and dict is a re-balanced red/black tree (satisfying all inv's)
      and same black height n.
   *)
(*[ restore_right <: 'a badRight -> 'a rbt ]*)
fun restore_right (Black(e:'a entry, Red lt, Red (rt as (_,Red _,_)))) =
       Red(e, Black lt, Black rt)     (* re-color *)
  | restore_right (Black(e, Red lt, Red (rt as (_,_,Red _)))) =
       Red(e, Black lt, Black rt)     (* re-color *)
  | restore_right (Black(e, l, Red(re, Red(rle, rll, rlr), rr))) =
         (* l is black, deep rotate *)
       Black(rle, Red(e, l, rll), Red(re, rlr, rr))
  | restore_right (Black(e, l, Red(re, rl, rr as Red _))) =
       Black(re, Red(e, l, rl), rr) (* l is black, shallow rotate *)
  | restore_right dict = dict

(*[    insert <: 'a rbt * 'a entry -> 'a rbt ]*)
fun 'a insert (dict, entry as (key,datum)) =
  let    (* ins (Red _) may violate color invariant at root
            ins (Black _) or ins (Empty) will be red/black tree
            ins preserves black height   *)
    (*[ ins <:  'a rbt -> 'a badRoot
              & 'a bt -> 'a rbt     ]*)
    fun ins (Empty) = Red(entry, Empty, Empty)
      | ins (Red(entry1 as (key1, datum1), left, right)) =
        (case compare(key,key1)
           of EQUAL => Red(entry, left, right)
            | LESS => Red(entry1, ins left, right)
            | GREATER => Red(entry1, left, ins right))
      | ins (Black(e1 as (key1, datum1), l, r)) =
        (case compare(key,key1)
           of EQUAL => Black(entry, l, r)
            | LESS => restore_left (Black(e1, ins l, r))
            | GREATER => restore_right (Black(e1, l, ins r)))
  in
    case ins dict
      of Red (t as (_, Red _, _)) => Black t   (* re-color *)
       | Red (t as (_, _, Red _)) => Black t   (* re-color *)
       | dict => dict     (* depend on sequential matching *)
  end
```

Figure 9.2: Functions for insertion into red-black trees

```
(*[ restore_right <: 'a badRight -> 'a rbt  ]*)
fun restore_right (Black(e, lt, Red (re, rlt, Red rrt))) =
       Red(re, Black (e, lt, rlt), Black rrt)
  | restore_right (Black(e, lt, Red (re, Red (rlte, rllt, rlrt), rrt))) =
       Red(rlte, Black (e, lt, rllt), Black (re, rlrt, rrt))
  | restore_right dict =
       dict
```

Figure 9.3: Okasaki's simplified rotations for red-black trees

```
(*
  norm.sml
  Author: Frank Pfenning
  Some normal-form invariants on lambda-calculus terms
  in deBruijn representation with explicit substitutions
*)

datatype term =
  Var of int
| Lam of term
| App of term * term;

datatype subst =
  Dot of term * subst
| Shift of int

(*[
(* Weak head-normal terms *)
datasort whnf = Lam of term | Var of int | App of head * term
     and head = Var of int | App of head * term

(* Normal terms *)
datasort norm = Lam of norm | Var of int | App of elim * norm
     and elim = Var of int | App of elim * norm
]*)

(*[ subst <: term * subst -> term  ]*)
(*[ comp <: subst * subst -> subst ]*)
fun subst (Var(1), Dot(e, s)) = e
  | subst (Var(n), Dot(e, s)) = (* n > 1 *)
      subst (Var(n-1), s)
  | subst (Var(n), Shift(k)) = Var(n+k)
  | subst (Lam(e), s) = Lam (subst (e, Dot(Var(1), comp(s, Shift(1)))))
  | subst (App(e1, e2), s) =
      App (subst (e1, s), subst (e2, s))
and comp (Shift(0), s') = s'
  | comp (Shift(n), Dot (e, s')) = comp (Shift (n-1), s')
  | comp (Shift(n), Shift(k)) = Shift(n+k)
  | comp (Dot(e, s), s') = Dot (subst (e, s'), comp (s, s'))
```

Figure 9.4: Normal forms experiment: Part One

```
(*[ whnf <: term -> whnf         ]*)
(*[ apply <: whnf * term -> whnf ]*)
fun whnf (Var(n)) = Var(n)
  | whnf (Lam(e)) = Lam(e)
  | whnf (App(e1,e2)) = apply (whnf e1, e2)
and apply (Var(n), e2) = App(Var(n), e2)
  | apply (Lam(e), e2) = whnf (subst (e, Dot(e2, Shift(0))))
  | apply (App(e11, e12), e2) = App(App(e11, e12), e2)

(*[ norm <: term -> norm           ]*)
(*[ appnorm <: norm * term -> norm ]*)
fun norm (Var(n)) = Var(n)
  | norm (Lam(e)) = Lam(norm e)
  | norm (App(e1,e2)) = appnorm (norm e1, e2)
and appnorm (Var(n), e2) = App(Var(n), norm e2)
  | appnorm (Lam(e), e2) = norm (subst (e, Dot(e2, Shift(0))))
  | appnorm (App(e11, e12), e2) = App(App(e11, e12), norm e2)

(*[ K <: norm ]*)
val K = Lam (Lam (Var 2))  (* \x.\y.x *)
(*[ S <: norm ]*)
val S = Lam (Lam (Lam (App (App (Var 3, Var 1), App (Var 2, Var 1)))))
                          (* \x.\y.\z. x z (y z) *)
(*[ I <: norm ]*)
val I = Lam (Var 1)        (* \x. x *)

(*[ ex1 <: norm ]*)
val ex1 = norm (App (I, I))  (* \x. x *)
(*[ ex2 <: norm ]*)
val ex2 = norm (App (App (S, K), K)) (* \x. x *)
```

Figure 9.5: Normal forms experiment: Part Two

and uses lists to represent stacks of unresolved operators and atomic values. When attempting to use datasort declarations to capture the invariants for these lists, we found ourselves wanting to define refinements of the type of operator lists, which is not possible. This an example of the problem with instantiations of datatypes described in Subsection 7.4.4. In this case we resolved the problem by replacing the use of the list type by a specialized datatype for lists of operators and atoms. This was quite easy in this case, since none of the standard functions for lists are used in this code.

We also found that we needed to provide sort annotations to correctly instantiate the sort of one polymorphic function. This is the function `error : region * string -> 'a` that raises an exception when an error is found in the input. Rather than annotate each application of this function, we found it convenient to define names for the three different instantiations that are required. While this was easy to do, it does suggest that a more sophisticated approach to parametric polymorphism is desirable in practice.

The datasort declarations for this example would have been slightly simpler if we added a feature that allowed one datasort to be defined as the union of some others. This should be relatively easy to implement, and we will consider adding this feature in the near future.

This was the largest of the experiments. Overall, it was a success: despite the complexity of the invariants, they were quite elegantly expressed using sorts, and checked using the sort checker. However, there were some minor undesirable outcomes as well: the need to modify the code to use a specialized datatype, the need to explicitly instantiate a polymorphic function, and the need to write out datasort declarations for unions in full, as described above. These indicate areas where improvements could be made, and such improvements will be considered in future work.

The datatype and datasort declarations appear in Figure 9.3. Various small functions appear in Figure 9.3. The function for performing a "shift" appears in Figure 9.3, and Figure 9.3 contains the main function for resolving an operator given a stack of unresolved operators and atoms.

9.4 Kaplan and Tarjan's purely functional lists

This experiment was performed by Kevin Watkins, and involved an implementation of a data structure for real-time purely functional lists based on the data structure of Kaplan and Tarjan [KT99]. This data structure supports efficient access and concatenation, and depends on some quite complicated invariants that appear to be suitable candidates for checking using datasort refinements.

Alas, a performance issue with the analysis of datasort declarations was encountered during this experiment, and could not be resolved with the version of the implementation available at that time, despite trying a number of different ways of formulating the invariants. This performance issue was only recently resolved by adding optimizations to the analysis of datasort declarations (see Section 8.4) hence the code has not yet been completed, although the basic stack operations and their invariants are implemented. Thus, this experiment must be considered at least a partial failure, although the resolution of the performance issue gives some optimism that with the current implementation the desired invariants could be successfully checked using sorts.

```
datatype 'a operator =    (* Operators and atoms for fixity parsing *)
    Atom of 'a
  | Infix of (FX.precedence * FX.associativity) * ('a * 'a -> 'a)
  | Prefix of FX.precedence * ('a -> 'a)
  | Postfix of FX.precedence * ('a -> 'a)

(*[  datasort 'a Atom = Atom of 'a
          and 'a Infix = Infix of (FX.precedence * FX.associativity) * ('a * 'a -> 'a)
          and 'a Prefix = Prefix of FX.precedence * ('a -> 'a)
          and 'a Postfix = Postfix of FX.precedence * ('a -> 'a);
     datasort 'a shiftable =
               Infix of (FX.precedence * FX.associativity) * ('a * 'a -> 'a)
            | Prefix of FX.precedence * ('a -> 'a)
            | Atom of 'a
]*)

type term = ExtSyn.term
type opr = term operator
(* type stack = (ExtSyn.term operator) list *) (* Replaced by the datatype below. *)

(* Specialized list datatype, so that we can define non-parametric refinements. *)
datatype stack = snil | ::: of opr * stack;
infixr 5 :::
                     (* Various refinements of stacks to enforce invariants *)
(*[ datasort pSnil = snil
         and pOp = ::: of term Infix * pAtom
                 | ::: of term Prefix * pSnil
                 | ::: of term Prefix * pOp
         and pAtom = ::: of term Atom * pSnil
                   | ::: of term Atom * pOp
    datasort pStable = snil    (* pOp | pAtom | pSnil *)
                     | ::: of term Atom * pSnil
                     | ::: of term Atom * pOp
                     | ::: of term Infix * pAtom
                     | ::: of term Prefix * pSnil
                     | ::: of term Prefix * pOp
    datasort pComplete = ::: of term Atom * pSnil
                       | ::: of term Atom * pOp
                       | ::: of term Postfix * pAtom
    datasort pRedex = ::: of term Postfix * pAtom
                    | ::: of term Atom * pOp
    datasort p = snil     (* pStable | pRedex *)
               | ::: of term Atom * pSnil
               | ::: of term Atom * pOp
               | ::: of term Infix * pAtom
               | ::: of term Postfix * pAtom
               | ::: of term Prefix * pSnil
               | ::: of term Prefix * pOp
]*)
```

Figure 9.6: Datatype and datasort declarations for fixity resolution

```
(* An instantiation of error to a particular sort *)
val serror = (Parsing.error (*[ <: Paths.region * string -> (pAtom & pSnil) ]*) )

(* The invariants were described as follows, prior to adding sorts. *)
(* Stack invariants, refinements of operator list *)
(*
   <p>       ::= <pStable> | <pRed>
   <pStable> ::= <pAtom> | <pOp?>
   <pAtom>   ::= Atom _ :: <pOp?>
   <pOp?>    ::= nil | <pOp>
   <pOp>     ::= Infix _ :: <pAtom> :: <pOp?>
               | Prefix _ :: <pOp?>
   <pRed>    ::= Postfix _ :: Atom _ :: <pOp?>
               | Atom _ :: <pOp>
*)
(* val reduce : <pRed> -> <p> *)
(*[ reduce <: pRedex -> pAtom ]*)
fun reduce (Atom(tm2):::Infix(_,con):::Atom(tm1):::p') =
       Atom(con(tm1,tm2)):::p'
  | reduce (Atom(tm):::Prefix(_,con):::p') = Atom(con(tm)):::p'
  | reduce (Postfix(_,con):::Atom(tm):::p') = Atom(con(tm)):::p'
  (* no other cases should be possible by stack invariant *)

(* val reduceRec : <pStable> -> ExtSyn.term *)
(*[ reduceRec <: pComplete -> term ]*)
fun reduceRec (Atom(e):::snil) = e
  | reduceRec (p) = reduceRec (reduce p)

(* val reduceAll : <p> -> ExtSyn.term *)
(*[ reduceAll <: Paths.region * p -> term ]*)
fun reduceAll (r, Atom(e):::snil) = e
  | reduceAll (r, Infix _:::p') = Parsing.error (r, "Incomplete infix expression")
  | reduceAll (r, Prefix _:::p') = Parsing.error (r, "Incomplete prefix expression")
  | reduceAll (r, snil) = Parsing.error (r, "Empty expression")
  | reduceAll (r, p) = reduceRec (reduce p)

(* val shiftAtom : term * <pStable> -> <p> *)
(* does not raise Error exception *)
(*[ shiftAtom <: term * pStable -> pStable ]*)
fun shiftAtom (tm, p as (Atom _:::p')) =   (* insert juxOp operator and reduce *)
      reduce (Atom(tm):::juxOp:::p)        (* juxtaposition binds most strongly *)
  | shiftAtom (tm, p) = Atom(tm):::p
```

Figure 9.7: Misc. small functions for fixity resolution

```
(* val shift : Paths.region * opr * <pStable> -> <p> *)
(*[ shift <:  Paths.region * term shiftable * pStable -> pStable
           & Paths.region * term Postfix * pAtom -> pRedex
           & Paths.region * term Postfix * pSnil -> pStable
           & Paths.region * term Postfix * pOp -> pStable    ]*)
fun shift (r, opr as Atom _, p as (Atom _:::p')) =  (* insert juxOp operator and reduce *)
      reduce (opr:::juxOp:::p)                      (* juxtaposition binds most strongly *)

  (* Atom/Infix: shift *)
  (* Atom/Prefix: shift *)
  (* Atom/Postfix cannot arise *)
  (* Atom/Empty: shift *)
  (* Infix/Atom: shift *)

  | shift (r, Infix _, Infix _:::p') =
      serror (r, "Consective infix operators")
  | shift (r, Infix _, Prefix _:::p') =
      serror (r, "Infix operator following prefix operator")

  (* Infix/Postfix cannot arise *)

  | shift (r, Infix _, snil) =
      serror (r, "Leading infix operator")
  | shift (r, opr as Prefix _, p as (Atom _:::p')) =   (* insert juxtaposition operator *)
      opr:::juxOp:::p                                  (* will be reduced later *)

  (* Prefix/Infix,Prefix,Empty: shift *)
  (* Prefix/Postfix cannot arise *)
  (* Postfix/Atom: shift, reduced immediately *)

  | shift (r, Postfix _, Infix _:::p') =
      serror (r, "Postfix operator following infix operator")
  | shift (r, Postfix _, Prefix _:::p') =
      serror (r, "Postfix operator following prefix operator")

  (* Postfix/Postfix cannot arise *)

  | shift (r, Postfix _, snil) =
      serror (r, "Leading postfix operator")
  | shift (r, opr, p) = opr:::p
```

Figure 9.8: Shift function for fixity resolution

```
(* val resolve : Paths.region * opr * <pStable> -> <p> *)
(* Decides, based on precedence of opr compared to the top of the
   stack whether to shift the new operator or reduce the stack
*)
(*[ resolve <: Paths.region * term operator * pStable -> pStable  ]*)
fun resolve (r, opr as Infix((prec, assoc), _), p as (Atom(_):::Infix((prec', assoc'), _):::p')) =
    (case (FX.compare(prec,prec'), assoc, assoc')
       of (GREATER,_,_) => shift(r, opr, p)
        | (LESS,_,_) => resolve (r, opr, reduce(p))
        | (EQUAL, FX.Left, FX.Left) => resolve (r, opr, reduce(p))
        | (EQUAL, FX.Right, FX.Right) => shift(r, opr, p)
        | _ => serror (r, "Ambiguous: infix after infix of same precedence"))

  | resolve (r, opr as Infix ((prec, assoc), _), p as (Atom(_):::Prefix(prec', _):::p')) =
    (case FX.compare(prec,prec')
       of GREATER => shift(r, opr, p)
        | LESS => resolve (r, opr, reduce(p))
        | EQUAL => serror (r, "Ambiguous: infix after prefix of same precedence"))

  (* infix/atom/atom cannot arise *)
  (* infix/atom/postfix cannot arise *)
  (* infix/atom/<empty>: shift *)

  (* always shift prefix *)
  | resolve (r, opr as Prefix _, p) =
      shift(r, opr, p)

  (* always reduce postfix, possibly after prior reduction *)
  | resolve (r, opr as Postfix(prec, _), p as (Atom _:::Prefix(prec', _):::p')) =
      (case FX.compare(prec,prec')
         of GREATER => reduce (shift (r, opr, p))
          | LESS => resolve (r, opr, reduce (p))
          | EQUAL => serror (r, "Ambiguous: postfix after prefix of same precedence"))

  (* always reduce postfix *)
  | resolve (r, opr as Postfix(prec, _), p as (Atom _:::Infix((prec', _), _):::p')) =
      (case FX.compare(prec,prec')
         of GREATER => reduce (shift (r, opr, p))
          | LESS => resolve (r, opr, reduce (p))
          | EQUAL => serror (r, "Ambiguous: postfix after infix of same precedence"))

  | resolve (r, opr as Postfix _, p as (Atom _:::snil)) =
      reduce (shift (r, opr, p))

  (* default is shift *)
  | resolve (r, opr, p) = shift(r, opr, p)
```

Figure 9.9: Resolve function for fixity resolution

The definitions of the types and sorts for this example appear in Figure 9.10. The code for the stack operations appears in Figure 9.11.

The performance issue arose while trying to extend the datasort declarations to double ended queues. This requires a number of changes, including the addition of a datatype for "paragraphs" which are lists of sentences. When the earlier version of the implementation attempted to check inclusion between some of the refinements of this datatype, it ran for longer than 12 hours without producing a result. With the current implementation the whole analysis takes only a few seconds (about 4.8 seconds on a 1.7GHz Pentium M). This is despite the fact that a lattice with 144 elements and over 10,000 intersections between elements is created.

```
datatype 'a bal = Leaf of 'a | Pair of 'a bal * 'a bal
infixr 3 $
infixr 2 $$
nonfix @ @@

(* A "digit" in this representation is a list (of length 0, 1, or 2)
   of perfectly balanced trees (each of which has the same depth)

   A "word" is a list of digits (of increasing depth)

   A "sentence" is a list of words

   The "word" and "sentence" structures exist just to give us pointers to various
   positions within the data structure where delayed work is waiting to be done.
   This interruption of the natural flow of the data structure is what makes the
   associated refinements complicated.  If we could state the refinements for the
   "telescoped" structure instead, and then describe the real structure as a
   sequence of "pieces" of the whole structure, where each "piece" fits into a hole
   in the next, it might be simpler.

   This is sort of analogous to the way a stack-machine based operational semantics
   can be stated in terms of a list of continuations, each fitting into the hole in
   the next one, with the types (here the refinements) having to mesh in the right way.
*)
datatype 'a word = @ | $ of 'a bal list * 'a word
datatype 'a sentence = @@ | $$ of 'a word * 'a sentence
type 'a stack = 'a sentence

(*[ datasort 'a zero = nil
    datasort 'a one = :: of 'a * 'a zero
    datasort 'a two = :: of 'a * 'a one

    datasort 'a nonempty_w = $ of 'a bal list * 'a word
    datasort 'a nonempty_s = $$ of 'a word * 'a sentence

    datasort 'a ones_w = @ | $ of 'a bal one * 'a ones_w
    datasort 'a zero_ones_w = $ of 'a bal zero * 'a ones_w
    datasort 'a two_ones_w = $ of 'a bal two * 'a ones_w

    datasort 'a two_s = @@ | $$ of 'a two_ones_w * 'a zero_s
    and      'a zero_s = $$ of 'a zero_ones_w * 'a two_s

    datasort 'a valid = $$ of 'a ones_w * 'a two_s

    datasort 'a nonempty = $$ of 'a nonempty_w * 'a two_s
                         | $$ of 'a ones_w * 'a nonempty_s
]*)
```

Figure 9.10: Datatype and datasort declarations for purely functional stacks

```
(*[ two_to_zero <: 'a two_s -> 'a zero_s ]*)
fun two_to_zero @@ = ([] $ @) $$ @@
  | two_to_zero (([x,y] $ [z] $ wd) $$ sn) =
      ([] $ @) $$ ([Pair(x,y),z] $ wd) $$ sn
  | two_to_zero (([x,y] $ @) $$ ([] $ wd) $$ sn) =
      ([] $ [Pair(x,y)] $ wd) $$ sn

(*[ push <: 'a * 'a valid -> ('a valid & 'a nonempty) ]*)
fun push (x, @ $$ sn) =
      let val ([] $ wd) $$ sn = two_to_zero sn
      in ([Leaf(x)] $ wd) $$ sn end
  | push (x, ([y] $ wd) $$ sn) =
      let val sn = two_to_zero sn
      in @ $$ ([Leaf(x),y] $ wd) $$ sn end

(* These cannot be validated without index refinements
   to keep track of the depths of the balanced trees within
   the structure. *)
fun unleaf (Leaf(x)) = x
  | unleaf _ = raise Match
fun unpair (Pair(x,y)) = (x,y)
  | unpair _ = raise Match

(*[ zero_to_two <: 'a zero_s -> 'a two_s ]*)
fun zero_to_two (([] $ @) $$ @@) = @@
  | zero_to_two (([] $ @) $$ ([xy,z] $ wd) $$ sn) =
      let val (x,y) = unpair xy
      in ([x,y] $ [z] $ wd) $$ sn end
  | zero_to_two (([] $ [xy] $ wd) $$ sn) =
      let val (x,y) = unpair xy
      in ([x,y] $ @) $$ ([] $ wd) $$ sn end

(*[ pop <: ('a valid & 'a nonempty) -> 'a * 'a valid ]*)
fun pop (([xx] $ wd) $$ sn) =
      let val x = unleaf xx
          val sn = zero_to_two (([] $ wd) $$ sn)
      in (x, @ $$ sn) end
  | pop (@ $$ ([xx,y] $ wd) $$ sn) =
      let val x = unleaf xx
          val sn = zero_to_two sn
      in (x, ([y] $ wd) $$ sn) end
```

Figure 9.11: Stack operations for purely functional lists

Chapter 10

Conclusion

We have demonstrated that sort checking using a bidirectional approach is practical for real programs, and that it allows the specification of many common properties in a natural way. We have done this by building a sort checker for Standard ML based on this approach, which we hope will be of use to programmers, and should at least allow significant experience to be gained with the features of our design in the context of real programming. Notable such features include subtyping, intersection types, a new form of value restriction on polymorphism, recursively defined refinements of datatypes, and bidirectional checking itself.

Our experiments so far have already suggested improvements to our design, but much of the future direction of this work depends on further practical experience with programming with sorts using our implementation. Potential future work has been outlined as appropriate in the preceding chapters. Notable such work includes considering a more principled approach to parametric polymorphism, along and the lines of that outlined in Section 7.5. We also plan to conduct further experiments, and to use the sort checker in all future projects involving SML programming in order to gain more experience with sort checking in practice.

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pp. 104–118; also DEC/Compaq Systems Research Center Research Report number 62, August 1990.

[AM91] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, June 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages*, pages 163–173, 1994.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[BDCdL95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, June 1995.

[BF99] Gilles Barthe and Maria João Frade. Constructor subtyping. In *Proceedings of ESOP'99*, volume 1576 of *LNCS*, pages 109–127, 1999.

[BH98] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

[Car84] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2:347–359, 1984.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CDG+] Hubert Common, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on *http://www.grappa.univ-lille3.fr/tata*.

[CDV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional character of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[CH03] James Cheney and Ralf Hinze. First-class phantom types. Technical Report Technical Report 1901, Cornell University, 2003.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Den98] Ewen Denney. Refinement types for specification. In David Gries and Willem-Paul de Roever, editors, *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 148–166, Shelter Island, New York, USA, 1998. Chapman and Hall.

[DG94] Razvan Diaconescu and Joseph Goguen. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[DP00] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming (ICFP'00)*, pages 198–208, Montreal, Canada, September 2000. ACM Press.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[DP03] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In A.D. Gordon, editor, *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, pages 250–266, Warsaw, Poland, April 2003. Springer-Verlag LNCS 2620.

[DP04] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In X.Leroy, editor, *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04)*, pages 281–292, Venice, Italy, January 2004. ACM Press. Extended version available as Technical Report CMU-CS-04-117, March 2004.

[Dun02] Joshua Dunfield. Combining two forms of type refinement. Technical Report CMU-CS-02-182, Carnegie Mellon University, September 2002.

[EP91] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[FCB02] Alain Frisch, Giuseppe Castagna, and Veronique Benzaken. Semantic subtyping. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[FFA99] Jeffrey S. Foster, Manuel Fhndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203. ACM Press, 1999.

[FFK+96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[Fil94] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM Symp. on Principles of Programming Languages (POPL'94)*, pages 446–457, Portland, Oregon, January 1994.

[Fil99] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM Symp. on Principles of Programming Languages (POPL'99)*, pages 175–188, San Antonio, Texas, 1999.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[FP02] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *Proceedings 2nd IFIP International Conference on Theoretical Computer Science (TCS'02)*, pages 448–460, 2002.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Gen69] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[GJ91] Carsten Gomard and Neil Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, pages 209–223. Springer-Verlag, 2002.

[Hay94] Susumu Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109:174–210, 1994.

[Hin69] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–40, 1969.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HS00] Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[Kah01] Stefan Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11(4):425–432, July 2001.

[KPT96] Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, January 1996.

[KT99] Haim Kaplan and Robert Endre Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46:577–603, 1999.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978.

[Mis84] Prakeesh Mishra. Towards a theory of types in prolog. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, 1984.

[Mit84] John Mitchell. Coercion and type inference (summary). In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[Mog89] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Computation*, 18(1/2):95–130, October/November 1986.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proc. of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–226, Uppsala, Sweden, September 2003.

[Nai04] Mayur Naik. A type system equivalent to a model checker. Master's thesis, Purdue University, 2004.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[NP04] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. Unpublished manuscript available from *http://www.cs.ucla.edu/~palsberg*, October 2004.

[NS95] H.R. Nielson and K.L. Solberg, editors. *Types for Program Analysis*, University of Aarhus, Denmark, May 1995. Informal Workshop Proceedings.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[Pau86] Lawrence Paulson. Natural deduction proof as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993. University of Nijmegen.

[Pfe01a] Frank Pfenning. Personal communication, 2001.

[Pfe01b] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[Pfe05] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2005. To appear. Draft from April 1997 available electronically.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1991.

[Pie97] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997. Summary in *Typed Lambda Calculi and Applications*, March 1993, pp. 346–360.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(4):576–599, 1995.

[PP01] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, 2001.

[PS98] Frank Pfenning and Carsten Schrmann. Twelf home page. Available at *http://www.cs.cmu.edu/~twelf*, 1998.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998. Full version to appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2000.

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[Rey69] John C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.

[Rey81] John C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 675–700. Springer-Verlag, September 1991.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[Rey02] John C. Reynolds. What do types mean? — From intrinsic to extrinsic semantics. In Annabelle McIver and Carroll Morgan, editors, *Essays on Programming Methodology*. Springer-Verlag, New York, 2002.

[Sei90] Hermut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[Ska97] Chris Skalka. Some decision problems for ML refinement types. Master's thesis, Carnegie-Mellon University, July 1997.

[Sol95] Kirsten Lackner Solberg. *Annotated Type Systems for Program Analysis*. PhD thesis, Odense University, Denmark, November 1995. Available from *http://www.daimi.au.dk/publications/PB/498/PB-498.pdf*.

[WC94] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *Proc 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, Orlando, FL, June 1994. ACM.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. Preliminary version is Rice Technical Report TR91-160.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Information and Computation*, 8:343–55, 1995.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235. ACM Press, 2003.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.