Typed Compilation of Recursive Datatypes

Joseph C. Vanderwaart Derek Dreyer Leaf Petersen
Karl Crary Robert Harper Perry Cheng

December 2002
CMU-CS-02-200

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Standard ML employs an opaque (or generative) semantics of datatypes, in which every datatype declaration produces a new type that is different from any other type, including other identically defined datatypes. A natural way of accounting for this is to consider datatypes to be abstract. When this interpretation is applied to type-preserving compilation, however, it has the unfortunate consequence that datatype constructors cannot be inlined, substantially increasing the run-time cost of constructor invocation compared to a traditional compiler. In this paper we examine two approaches to eliminating function call overhead from datatype constructors. First, we consider a transparent interpretation of datatypes that does away with generativity, altering the semantics of SML; and second, we propose an interpretation of datatype constructors as coercions, which have no run-time effect or cost and faithfully implement SML semantics.

Support for this research was provided by the National Science Foundation through grants CCR-9984812, “Type-Driven Technology for Software and Information Infrastructure,” and CCR-0121633, “Language Technology for Trustless Software Dissemination,” and the Advanced Research Projects Agency CSTO through contract F19628-95-C-0050, “The Fox Project: Advanced Languages for System Software.” The views and conclusions in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of these agencies or the U.S. Government.

Keywords: Typed compilation, Standard ML, recursive types, coercions

1 Introduction

The programming language Standard ML (SML) [9] provides a distinctive mechanism for defining recursive types, known as a *datatype declaration.* For example, the following declaration defines the type of lists of integers:

```
datatype intlist = Nil
                 | Cons of int * intlist
```

This datatype declaration introduces the type `intlist` and two *constructors*: `Nil` represents the empty list, and `Cons` combines an integer and a list to produce a new list. For instance, the expression `Cons (1, Cons (2, Cons (3,Nil)))` has type `intlist` and corresponds to the list $[1, 2, 3]$. Values of this datatype are deconstructed by a case analysis that examines a list and determines whether it was constructed with `Nil` or with `Cons`, and in the latter case, extracting the original integer and list.

An important aspect of SML datatypes is that they are *generative.* That is, every datatype declaration defines a type that is distinct from any other type, including those produced by other, possibly identical, datatype declarations. The formal Definition of SML [9] makes this precise by stating that a datatype declaration produces a new type name, but does not associate that name with a definition; in this sense, datatypes are similar to abstract types. Harper and Stone [7] (hereafter, HS) give a type-theoretic interpretation of SML by exhibiting a translation from SML into a simpler, typed *internal language.* This translation is faithful to the Definition of SML in the sense that, with a few well-known exceptions, it translates an SML program into a well-typed IL program if and only if the SML program is well-formed according to the Definition. Consequently, we consider HS to be a suitable foundation for type-directed compilation of SML. Furthermore, it seems likely that any other suitable type-theoretic interpretation (*i.e.,* one that is faithful to the Definition) will encounter the same issues we explore in our analysis.

Harper and Stone capture datatype generativity by translating a datatype declaration as a module containing an abstract type and functions to construct and deconstruct values of that type; thus in the setting of the HS interpretation, datatypes *are* abstract types. The generativity of datatypes poses some challenges for type-directed compilation of SML. In particular, although the HS interpretation is easy to understand and faithful to the Definition of SML, it is inefficient when implemented naïvely. The problem is that construction and deconstruction of datatype values require calls to functions exported by the module defining the datatype; this is unacceptable given the ubiquity of datatypes in SML code. Conventional compilers, which disregard type information after an initial type-checking phase, may dispense with this cost by *inlining* those functions; that is, they may replace the function calls with the actual code of the corresponding functions to eliminate the call overhead. A type-directed compiler, however, does not have this option since all optimizations, including inlining, must be type-preserving. Moving the implementation of a datatype constructor across the module boundary violates type abstraction and thus results in ill-typed intermediate code. This will be made more precise in Section 2.

In this paper, we will discuss two potential ways of handling this performance problem. We will present these alternatives in the context of the TILT/ML compiler developed at CMU [11, 14]; they are relevant, however, not just to TILT, but to understanding the definition of the language and type-preserving compilation in general.

The first approach is to do away with datatype generativity altogether, replacing the abstract types in the HS interpretation with concrete ones. We call this approach the *transparent interpretation of datatypes.* Clearly, a compiler that does this is *not* an implementation of Standard ML, and we will show that, although the modified language does admit inlining of datatype constructors, it has some unexpected properties. In particular, it is *not* the case that every well-formed SML program is allowed under the transparent interpretation.

In contrast, the second approach, which we have adopted in the most recent version of the TILT compiler, offers an efficient way of implementing datatypes in a typed setting that is consistent with the Definition. In particular, since a value of recursive type is typically represented at run time in the same way as its unrolling, we can observe that the mediating functions produced by the HS interpretation all behave like the identity function at run time. We replace these functions with special values that are distinguished from ordinary functions by the introduction of "coercion types". We call this the *coercion interpretation of datatypes*, and

$$\begin{array}{lrcl}
\textit{Types} & \sigma,\tau & ::= & \cdots \mid \alpha \mid \delta \\
\textit{Recursive Types} & \delta & ::= & \mu_i(\alpha_1,\ldots,\alpha_n).(\tau_1,\ldots,\tau_n) \\
\textit{Terms} & e & ::= & \cdots \mid x \mid \mathtt{roll}_\delta(e) \\
 & & & \mid \mathtt{unroll}_\delta(e) \\
\textit{Typing Contexts} & \Gamma & ::= & \epsilon \mid \Gamma, x : \tau \mid \Gamma, \alpha
\end{array}$$

Figure 1: Syntax of Iso-recursive Types

$$\begin{array}{rcl}
\vec{X} & \overset{\text{def}}{=} & X_1,\ldots,X_n \text{ for some } n \geq 1, \\
 & & \text{where } X \text{ is a metavariable,} \\
 & & \text{such as } \alpha \text{ or } \tau \\
\mathsf{length}(\vec{X}) & \overset{\text{def}}{=} & n, \text{ where } \vec{X} = X_1,\ldots,X_n \\
\mu\alpha.\,\tau & \overset{\text{def}}{=} & \mu_1(\alpha).(\tau) \\
\vec{\mu}(\vec{\alpha}).(\vec{\tau}) & \overset{\text{def}}{=} & \mu_1(\vec{\alpha}).(\vec{\tau}),\ldots,\mu_n(\vec{\alpha}).(\vec{\tau}), \\
 & & \text{where } \mathsf{length}(\vec{\alpha}) = \mathsf{length}(\vec{\tau}) = n \\
\mathsf{expand}(\delta) & \overset{\text{def}}{=} & \tau_i[\vec{\mu}(\vec{\alpha}).(\vec{\tau})/\vec{\alpha}],\ \text{where } \delta = \mu_i(\vec{\alpha}).(\vec{\tau})
\end{array}$$

Figure 2: Shorthand Definitions

argue that it allows a compilation strategy that generates code with a run-time efficiency comparable to what would be attained if datatype constructors were inlined.

The paper is structured as follows: Section 2 gives the details of the HS interpretation of datatypes (which we also refer to as the *opaque interpretation of datatypes*) and illustrates the problems with inlining. Section 3 discusses the transparent interpretation. Section 4 gives the coercion interpretation and discusses its properties. Section 5 gives a performance comparison of the three interpretations. Section 6 discusses related work and Section 7 concludes.

2 The Opaque Interpretation of Datatypes

In this section, we review the parts of Harper and Stone's interpretation of SML that are relevant to our discussion of datatypes. In particular, after defining the notation we use for our internal language, we will give an example of the HS elaboration of datatypes. We will refer to this example throughout the paper. We will also review the way Harper and Stone define the matching of structures against signatures, and discuss the implications this has for datatypes. This will be important in Section 3, where we show some differences between signature matching in SML and signature matching under our transparent interpretation of datatypes.

2.1 Notation

Harper and Stone give their interpretation of SML as a translation, called *elaboration*, from SML into a typed internal language (IL). We will not give a complete formal description of the internal language we use in this paper; instead, we will use ML-like syntax for examples and employ the standard notation for function, sum and product types. For a complete discussion of elaboration, including a thorough treatment of the internal language, we refer the reader to Harper and Stone [7]. Since we are focusing our attention on datatypes, *recursive types* will be of particular importance. We will therefore give a precise description of the semantics of the form of recursive types we use.

The syntax for recursive types is given in Figure 1. Recursive types are separated into their own syntactic subcategory, ranged over by δ. This is mostly a matter of notational convenience, as there are many times when we wish to make it clear that a particular type is a recursive one. A recursive type has the form

$\Gamma \vdash \text{ok}$	Well-formed context.
$\Gamma \vdash \tau \ \text{type}$	Well-formed type.
$\Gamma \vdash \sigma \equiv \tau$	Equivalence of types.
$\Gamma \vdash e : \tau$	Well-formed term.

Figure 3: Relevant Typing Judgments

$$\frac{i \in 1..n \qquad \forall j \in 1..n.\ \Gamma, \alpha_1, \ldots, \alpha_n \vdash \tau_j \ \text{type}}{\Gamma \vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n) \ \text{type}}$$

$$\frac{i \in 1..n \qquad \forall j \in 1..n.\ \Gamma, \alpha_1, \ldots, \alpha_n \vdash \sigma_j \equiv \tau_j}{\Gamma \vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\sigma_1, \ldots, \sigma_n) \equiv \mu_i(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n)}$$

$$\frac{\Gamma \vdash e : \mathsf{expand}(\delta)}{\Gamma \vdash \mathtt{roll}_\delta(e) : \delta} \qquad \frac{\Gamma \vdash e : \delta}{\Gamma \vdash \mathtt{unroll}_\delta(e) : \mathsf{expand}(\delta)}$$

Figure 4: Typing Rules for Iso-recursive Types

$\mu_i(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n)$, where $1 \leq i \leq n$ and each α_j is a type variable that may appear free in any or all of $\tau_1, \ldots, \tau_n$. Intuitively, this type is the ith in a system of n mutually recursive types. As such, it is isomorphic to τ_i with each α_j replaced by the jth component of the recursive bundle. Formally, it is isomorphic to the following somewhat unwieldy type:

$$\begin{aligned}&\tau_i[\mu_1(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n),\\ &\quad \ldots, \mu_n(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n)/\alpha_1, \ldots, \alpha_n]\end{aligned}$$

(where, as usual, we denote by $\tau[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]$ the simultaneous capture-avoiding substitution of $\sigma_1, \ldots, \sigma_n$ for $\alpha_1, \ldots, \alpha_n$ in τ). Since we will be writing such types often, we use some notational conventions to make things clearer; these are shown in Figure 2. Using these shorthands, the above type may be written as $\mathsf{expand}(\mu_i(\vec{\alpha}).(\vec{\tau}))$.

The judgment forms of the static semantics of our internal language are given in Figure 3, and the rules relevant to recursive types are given in Figure 4. Note that the only rule that can be used to judge two recursive types equal requires that the two types in question are the same (ith) projection from bundles of the same length whose respective components are all equal. In particular, there is no "unrolling" rule stating that $\delta \equiv \mathsf{expand}(\delta)$; type theories in which this equality holds are said to have *equi-recursive* types and are significantly more complex [5]. The recursive types in our theory are *iso-recursive* types that are isomorphic, but not equal, to their expansions. The isomorphism is embodied by the `roll` and `unroll` operations at the term level; the former turns a value of type $\mathsf{expand}(\delta)$ into one of type δ, and the latter is its inverse.

2.2 Elaborating Datatype Declarations

The HS interpretation of SML includes a full account of datatypes, including generativity. The main idea is to encode datatypes as recursive sum types but hide this implementation behind an opaque signature. A datatype declaration therefore elaborates to a structure that exports a number of abstract types and functions that construct and deconstruct values of those types. For example, consider the following pair of mutually recursive datatypes, representing expressions and declarations in the abstract syntax of a toy language:

```
datatype exp = VarExp of var
             | LetExp of dec * exp
     and dec = ValDec of var * exp
             | SeqDec of dec * dec
```

```
structure ExpDec :> sig
   type exp
   type dec
   val exp_in : var + (dec * exp) -> exp
   val exp_out : exp -> var + (dec * exp)
   val dec_in : (var * exp) +
                  (dec * dec) -> dec
   val dec_out : dec -> (var * exp) +
                          (dec * dec)
end = struct
   type exp = $\mu_1(\alpha,\beta)$.(var + $\beta$ * $\alpha$, var * $\alpha$ + $\beta$ * $\beta$)
   type dec = $\mu_2(\alpha,\beta)$.(var + $\beta$ * $\alpha$, var * $\alpha$ + $\beta$ * $\beta$)
   fun exp_in x = roll$_{exp}(x)$
   fun exp_out x = unroll$_{exp}(x)$
   fun dec_in x = roll$_{dec}(x)$
   fun dec_out x = unroll$_{dec}(x)$
end
```

Figure 5: Harper-Stone Elaboration of `exp-dec` Example

The HS elaboration of this declaration is given in Figure 5, using ML-like syntax for readability. To construct a value of one of these datatypes, a program must use the corresponding `in` function; these functions each take an element of the sum type that is the "unrolling" of the datatype and produce a value of the datatype. More concretely, we implement the constructors for `exp` and `dec` as follows:

$$\begin{aligned}
\texttt{VarExp(x)} &\overset{\text{def}}{=} \texttt{ExpDec.exp\_in}(\texttt{inj}_1(\texttt{x}))\\
\texttt{LetExp(d,e)} &\overset{\text{def}}{=} \texttt{ExpDec.exp\_in}(\texttt{inj}_2(\texttt{d,e}))\\
\texttt{ValDec(x,e)} &\overset{\text{def}}{=} \texttt{ExpDec.dec\_in}(\texttt{inj}_1(\texttt{x,e}))\\
\texttt{SeqDec(d1,d2)} &\overset{\text{def}}{=} \texttt{ExpDec.dec\_in}(\texttt{inj}_2(\texttt{d1,d2}))
\end{aligned}$$

Notice that the types `exp` and `dec` are held abstract by the opaque signature ascription. This captures the generativity of datatypes, since the abstraction prevents `ExpDec.exp` and `ExpDec.dec` from being judged equal to any other types. However, as we mentioned in Section 1, this abstraction also prevents inlining of the `in` and `out` functions: for example, if we attempt to inline `exp_in` in the definition of `VarExp` above, we get

$$\texttt{VarExp(x)} \overset{\text{def}}{=} \texttt{roll}_{\texttt{ExpDec.exp}}(\texttt{inj}_1(\texttt{x}))$$

but this is ill-typed outside of the `ExpDec` module because the fact that `exp` is a recursive type is not visible. Thus performing inlining on well-typed code can lead to ill-typed code, so we say that inlining across abstraction boundaries is *not type-preserving* and therefore not an acceptable strategy for a typed compiler. The problem is that since we cannot inline `in` and `out` functions, our compiler must pay the run-time cost of a function call every time a value of a datatype is constructed or case-analyzed. Since these operations occur very frequently in SML code, this performance penalty is significant.

One strategy that can alleviate this somewhat is to hold the implementation of a datatype abstract *during elaboration*, but to expose its underlying implementation *after elaboration* to other code defined in the same compilation unit. Calls to the constructors of a locally-defined datatype can then be safely inlined. In the setting of whole-program compilation, this approach can potentially eliminate constructor call overhead for all datatypes except those appearing as arguments to functors. However, in the context of separate compilation, the clients of a datatype generally do not have access to its implementation, but rather only to the specifications of its constructors. As we shall see in Section 3, the specifications of a datatype's constructors do not provide sufficient information to correctly predict how the datatype is actually implemented, so the above compilation strategy will have only limited success in a true separate compilation setting.

2.3 Datatypes and Signature Matching

Standard ML makes an important distinction between datatype *declarations*, which appear at the top level or in structures, and datatype *specifications*, which appear in signatures. As we have seen, the HS interpretation elaborates datatype declarations as opaquely sealed structures; datatype specifications are translated into specifications of structures. For example, the signature

```
signature S = sig
  datatype intlist = Nil
                   | Cons of int * intlist
end
```

contains a datatype specification, and elaborates as follows:

```
signature S = sig
  structure Intlist : sig
     type intlist
     val intlist_in :
           unit + int * intlist -> intlist
     val intlist_out :
           intlist -> unit + int * intlist
  end
end
```

A structure M will match `S` if M contains a structure `Intlist` of the appropriate signature.[1] In particular, it is clear that the structure definition produced by the HS interpretation for the datatype `intlist` defined in Section 1 has this signature, so that datatype declaration matches the specification above.

What is necessary in general for a datatype declaration to match a specification under this interpretation? Since datatype declarations are translated as opaquely sealed structures, and datatype specifications are translated as structure specifications, matching a datatype declaration against a spec boils down to matching one signature—the one opaquely sealing the declaration structure—against another signature.

Suppose we wish to know whether the signature `S` matches the signature `T`; that is, whether a structure with signature `S` may also be given the signature `T`. Intuitively, we must make sure that for every specification in `T` there is a specification in `S` that is compatible with it. For instance, if `T` contains a value specification of the form `val x :` τ, then `S` must also contain a specification `val x :` τ', where $\tau' \equiv \tau$. For an abstract type specification of the form `type t` occurring in `T`, we must check that a specification of `t` also appears in `S`; furthermore, if the specification in `S` is a transparent one, say `type t =` τ_{imp}, then when checking the remainder of the specifications in `T` we may assume in both signatures that `t` $= \tau_{imp}$. Transparent type specifications in `T` are similar, but there is the added requirement that if the specification in `T` is `type t =` τ_{spec} and the specification in `S` is `type t =` τ_{imp}, then τ_{spec} and τ_{imp} must be equivalent.

Returning to the specific question of datatype matching, a specification of the form

```
datatype $t_1$ = $\tau_1$ and ... and $t_n$ = $\tau_n$
```

(where the τ_i may be sum types) elaborates to a specification of a structure with the following signature:

```
sig
     type $t_1$
     ...
     type $t_n$
     val $t_1$_in : $\tau_1$ -> $t_1$
     val $t_1$_out : $t_1$ -> $\tau_1$
     ...
     val $t_n$_in : $\tau_n$ -> $t_n$
     val $t_n$_out : $t_n$ -> $\tau_n$
end
```

[1] Standard ML allows only datatypes to match datatype specifications, so the actual HS elaboration must use a name for the datatype that cannot be guessed by a programmer.

```
structure ExpDec :> sig
   type exp = μ1(α,β).(var + β * α, var * α + β * β)
   type dec = μ2(α,β).(var + β * α, var * α + β * β)
   (* ... specifications for in and out functions
          same as before ... *)
end =
   (* ... same structure as before ... *)
```

Figure 6: The Transparent Elaboration of Exp and Dec

In order to match this signature, the structure corresponding to a datatype declaration must define types named $\mathtt{t}_1, \ldots, \mathtt{t}_n$ and must contain in and out functions of the appropriate type for each. (Note that in any structure produced by elaborating a datatype declaration under this interpretation, the $\mathtt{t}_i$'s will be abstract types.) Thus, for example, if $m \geq n$ then the datatype declaration

$\mathtt{datatype\ t}_1 = \sigma_1\ \mathtt{and}\ \ldots\ \mathtt{and\ t}_m = \sigma_m$

matches the above specification if and only if $\sigma_i \equiv \tau_i$ for $1 \leq i \leq n$, since this is necessary and sufficient for the types of the in and out functions to match for the types mentioned in the specification.

3 A Transparent Interpretation of Datatypes

A natural approach to enabling the inlining of datatypes in a type-preserving compiler is to do away with the generative semantics of datatypes. In the context of the HS interpretation, this corresponds to replacing the abstract type specification in the signature of a datatype module with a transparent type definition, so we call this modified interpretation the *transparent interpretation of datatypes* (TID).

3.1 Making Datatypes Transparent

The idea of the transparent interpretation is to expose the implementation of datatypes as recursive sum types during elaboration, rather than hiding it. In our expdec example, this corresponds to changing the declaration shown in Figure 5 to that shown in Figure 6 (we continue to use ML-like syntax for readability).

Importantly, this change must extend to datatype specifications as well as datatype declarations. Thus, a structure that exports a datatype must export its implementation transparently, using a signature similar to the one in the figure—otherwise a datatype inside a structure would appear to be generative outside that structure, and there would be little point to the new interpretation.

As we have mentioned before, altering the interpretation of datatypes to expose their implementation as recursive types really creates a new language, which is neither a subset nor a superset of Standard ML. An example of the most obvious difference can be seen in Figure 7. In the figure, two datatypes are defined by seemingly identical declarations. In SML, because datatypes are generative, the two types List1.t and List2.t are distinct; since the variable l has type List1.t but is passed to List2.Cons, which expects List2.t, the function switch is ill-typed. Under the transparent interpretation, however, the implementations of both datatypes are exported transparently as $\mu\alpha.\mathtt{unit} + \mathtt{int}\ \mathtt{*}\ \alpha$. Thus under this interpretation, List1.t and List2.t are equal and so switch is a well-typed function.

It is clear that many programs like this one fail to type-check in SML but succeed under the transparent interpretation; what is less obvious is that there are some programs for which the opposite is true. We will discuss two main reasons for this.

3.2 Problematic Datatype Matchings

Recall that according to the HS interpretation, a datatype matches a datatype specification if the types of the datatype's in and out functions match the types of the in and out functions in the specification. (Note: the types of the out functions match if and only if the types of the in functions match, so we will

```
structure List1 = struct
     datatype t = Nil | Cons of int * t
end

structure List2 = struct
     datatype t = Nil | Cons of int * t
end

fun switch List1.Nil = List2.Nil
  | switch (List1.Cons (n,l)) =
    List2.Cons (n,l)
```

Figure 7: Non-generativity of Transparent Datatypes

hereafter refer only to the `in` functions.) Under the transparent interpretation, however, it is also necessary that the recursive type implementing the datatype match the one given in the specification. This is not a trivial requirement; we will now give two examples of matchings that succeed in SML but fail under the transparent interpretation.

3.2.1 A Simple Example

A very simple example of a problematic matching is the following. Under the opaque interpretation, matching the structure

```
struct
     datatype u = A of u * u | B of int
     type v = u * u
end
```

against the signature

```
sig
     type v
     datatype u = A of v | B of int
end
```

amounts to checking that the type of the `in` function for u defined in the structure matches that expected by the signature once `u * u` has been substituted for v in the signature. (No definition is substituted for u, since it is abstract in the structure.) After substitution, the type required by the signature for the `in` function is `u * u + int -> u`, which is exactly the type of the function given by the structure, so the matching succeeds.

Under the transparent interpretation, however, the structure defines u to be $\mathtt{u}_{imp} \overset{\text{def}}{=} \mu\alpha.\, \alpha \mathbin{\mathtt{*}} \alpha \mathbin{\mathtt{+}} \mathtt{int}$ but the signature specifies u as $\mu\alpha.\, v \mathbin{\mathtt{+}} \mathtt{int}$. In order for matching to succeed, these two types must be equivalent after we have substituted $\mathtt{u}_{imp} \mathbin{\mathtt{*}} \mathtt{u}_{imp}$ for v in the specification. That is, it is required that

$$\mathtt{u}_{imp} \equiv \mu\alpha.\, \mathtt{u}_{imp} \mathbin{\mathtt{*}} \mathtt{u}_{imp} \mathbin{\mathtt{+}} \mathtt{int}$$

Observe that the type on the right is none other than $\mu\alpha.\, \mathsf{expand}(\mathtt{u}_{imp})$. (Notice also that the bound variable α does not appear free in the body of this μ-type. Hereafter we will write such types with a wildcard _ in place of the type variable to indicate that it is not used in the body of the μ.) This equivalence does not hold for iso-recursive types, so the matching fails.

3.2.2 A More Complex Example

Another example of a datatype matching that is legal in SML but fails under the transparent interpretation can be found by reconsidering our running example of `exp` and `dec`. Under the opaque interpretation, a structure containing this pair of datatypes matches the following signature, which hides the fact that `exp` is a datatype:

```
sig
     type exp
     datatype dec = ValDec of var * exp
                  | SeqDec of dec * dec
end
```

When this datatype specification is elaborated under the transparent interpretation, however, the resulting IL signature looks like:

```
sig
     type exp
     type dec = $\mathtt{dec}_{spec}$
     ⋮
end
```

where $\mathtt{dec}_{spec} \overset{\text{def}}{=} \mu\alpha.\,\mathtt{var}\ \mathtt{*}\ \mathtt{exp}\ \mathtt{+}\ \alpha\ \mathtt{*}\ \alpha$. Elaboration of the declarations of `exp` and `dec`, on the other hand, produces the structure in Figure 6, which has the signature:

```
sig
     type exp = $\mathtt{exp}_{imp}$
     type dec = $\mathtt{dec}_{imp}$
     ⋮
end
```

where we define

$$\begin{aligned}\mathtt{exp}_{imp} &\overset{\text{def}}{=} \mu_1(\alpha,\beta).(\mathtt{var}\ \mathtt{+}\ \beta\ \mathtt{*}\ \alpha,\ \mathtt{var}\ \mathtt{*}\ \alpha\ \mathtt{+}\ \beta\ \mathtt{*}\ \beta)\\ \mathtt{dec}_{imp} &\overset{\text{def}}{=} \mu_2(\alpha,\beta).(\mathtt{var}\ \mathtt{+}\ \beta\ \mathtt{*}\ \alpha,\ \mathtt{var}\ \mathtt{*}\ \alpha\ \mathtt{+}\ \beta\ \mathtt{*}\ \beta)\end{aligned}$$

Matching the structure containing the datatypes against the signature can only succeed if $\mathtt{dec}_{spec} \equiv \mathtt{dec}_{imp}$ (under the assumption that $\mathtt{exp} \equiv \mathtt{exp}_{imp}$). This equivalence does not hold because the two μ-types have different numbers of components.

3.3 Problematic Signature Constraints

The module system of SML provides two ways to express sharing of type information between structures. The first, `where type`, modifies a signature by "patching in" a definition for a type the signature originally held abstract. The second, `sharing type`, asserts that two or more type names (possibly in different structures) refer to the same type. Both of these forms of constraints are restricted so that multiple inconsistent definitions are not given to a single type name. In the case of `sharing type`, for example, it is required that all the names be *flexible*, that is, they must either be abstract or defined as equal to another type that is abstract. Under the opaque interpretation, datatypes are abstract and therefore flexible, meaning they can be shared; under the transparent interpretation, datatypes are concretely defined and hence can never be shared. For example, the following signature is legal in SML:

```
signature S = sig
    structure M : sig
        type s
        datatype t = A | B of s
    end
    structure N : sig
        type s
        datatype t = A | B of s
    end
    sharing type M.t = N.t
end
```

We can write an equivalent signature by replacing the `sharing type` line with `where type t = M.t`, which is also valid SML. Neither of these signatures elaborates successfully under the transparent interpretation of datatypes, since under that interpretation the datatypes are transparent and therefore ineligible for either sharing or `where type`.

Another example is the following signature:

```
signature AB = sig
     structure A : sig
          type s
          val C : s
     end
     structure B : sig
          datatype t = C | D of A.s * t
     end
     sharing type A.s = B.t
end
```

(Again, we can construct an analogous example with `where type`.) Since the name `B.t` is flexible under the opaque interpretation but not the transparent, this code is legal SML but must be rejected under the transparent interpretation.

3.4 Relaxing Recursive Type Equivalence

We will now describe a way of weakening type equivalence (*i.e.*, making it equate more types) so that the problematic datatype matchings described in Section 3.2 succeed under the transparent interpretation. (This will not help with the problematic sharing constraints of Section 3.3.) The ideas in this section are based upon the equivalence algorithm adopted by Shao [8] for the FLINT/ML compiler.

To begin, consider the simple *u*-*v* example of Section 3.2.1. Recall that in that example, matching the datatype declaration against the spec required proving the equivalence

$$\mathtt{u}_{imp} \equiv \mu\alpha.\, \mathtt{u}_{imp} \mathbin{\texttt{*}} \mathtt{u}_{imp} \mathbin{\texttt{+}} \mathtt{int}$$

where the type on the right-hand side is just $\mu\,\_\,.\,\mathsf{expand}(\mathtt{u}_{imp})$. By simple variations on this example, it is easy to show that in general, for the transparent interpretation to be as permissive as the opaque, the following recursive type equivalence must hold:

$$\delta \equiv \mu\,\_\,.\,\mathsf{expand}(\delta)$$

We refer to this as the *boxed-unroll* rule. It says that a recursive type is equal to its unrolling "boxed" by a μ. An alternative formulation, equivalent to the first one by transitivity, makes two recursive types equal if their unrollings are equal, *i.e.*:

$$\frac{\mathsf{expand}(\delta_1) \equiv \mathsf{expand}(\delta_2)}{\delta_1 \equiv \delta_2}$$

Intuitively, this rule is needed because datatype matching succeeds under the opaque interpretation whenever the unrolled form of the datatype implementation equals the unrolled form of the datatype spec (because these are both supposed to describe the domain of the `in` function).

Although the boxed-unroll equivalence is necessary for the transparent interpretation of datatypes to admit all matchings admitted by the opaque one, it is not sufficient; to see this, consider the problematic `exp-dec` matching from Section 3.2.2. The problematic constraint in that example is:

$$\mathtt{dec}'_{spec} \equiv \mathtt{dec}_{imp}$$

where $\mathtt{dec}'_{spec} = \mathtt{dec}_{spec}[\mathtt{exp}_{imp}/\mathtt{exp}]$ (substituting $\mathtt{exp}_{imp}$ for `exp` in $\mathtt{dec}_{imp}$ has no effect, since the variable does not appear free). Expanding the definitions of these types, we get the constraint:

$$\begin{aligned}&\mu\alpha.\,\mathtt{var} \mathbin{\mathtt{*}} \mathtt{exp}_{imp} \mathbin{\mathtt{+}} \alpha \mathbin{\mathtt{*}} \alpha \equiv \\ &\quad \mu_2(\alpha,\beta).(\mathtt{var} \mathbin{\mathtt{+}} \beta \mathbin{\mathtt{*}} \alpha,\ \mathtt{var} \mathbin{\mathtt{*}} \alpha \mathbin{\mathtt{+}} \beta \mathbin{\mathtt{*}} \beta)\end{aligned}$$

The boxed-unroll rule is insufficient to prove this equivalence. In order to apply boxed-unroll to prove these two types equivalent, we must be able to prove that their unrollings are equivalent, in other words that

$$\begin{aligned}&\mathtt{var} \mathbin{\mathtt{*}} \mathtt{exp}_{imp} \mathbin{\mathtt{+}} \mathtt{dec}'_{spec} \mathbin{\mathtt{*}} \mathtt{dec}'_{spec} \equiv \\ &\quad \mathtt{var} \mathbin{\mathtt{*}} \mathtt{exp}_{imp} \mathbin{\mathtt{+}} \mathtt{dec}_{imp} \mathbin{\mathtt{*}} \mathtt{dec}_{imp}\end{aligned}$$

But we cannot prove this without first proving $\mathtt{dec}'_{spec} \equiv \mathtt{dec}_{imp}$, which is exactly what we set out to prove in the first place! The boxed-unroll rule is therefore unhelpful in this case.

The trouble is that proving the premise of the boxed-unroll rule (the equivalence of $\mathsf{expand}(\delta_1)$ and $\mathsf{expand}(\delta_2)$) may require proving the conclusion (the equivalence of δ_1 and δ_2). Similar problems have been addressed in the context of general equi-recursive types. In that setting, deciding type equivalence involves assuming the conclusions of equivalence rules when proving their premises [1, 2]. Applying this idea provides a natural solution to the problem discussed in the previous section. We can maintain a "trail" of type-equivalence assumptions; when deciding the equivalence of two recursive types, we add that equivalence to the trail before comparing their unrollings.

Formally, the equivalence judgement itself becomes $\Gamma; A \vdash \sigma \equiv \tau$, where A is a set of assumptions, each of the form $\tau_1 \equiv \tau_2$. All the equivalence rules in the static semantics must be modified to account for the trail. In all the rules except those for recursive types, the trail is simply passed unchanged from the conclusions to the premises. There are two new rules that handle recursive types:

$$\frac{\tau_1 \equiv \tau_2 \in A}{\Gamma; A \vdash \tau_1 \equiv \tau_2}$$

$$\frac{\Gamma; A \cup \{\delta_1 \equiv \delta_2\} \vdash \mathsf{expand}(\delta_1) \equiv \mathsf{expand}(\delta_2)}{\Gamma; A \vdash \delta_1 \equiv \delta_2}$$

The first rule allows an assumption from the trail to be used; the second rule is an enhanced form of the boxed-unroll rule that adds the conclusion to the assumptions of the premise. It is clear that the trail is just what is necessary in order to resolve the `exp-dec` anomaly described above; before comparing the unrollings of $\mathtt{dec}_{spec}$ and $\mathtt{dec}_{imp}$, we add the assumption $\mathtt{dec}_{spec} \equiv \mathtt{dec}_{imp}$ to the trail; we then use this assumption to avoid the cyclic dependency we encountered before.

In fact, the trailing version of the boxed-unroll rule is sufficient to ensure that the transparent interpretation accepts all datatype matchings accepted by SML. To see why, consider a datatype specification

$$\mathtt{datatype\ t}_1\ \mathtt{=}\ \tau_1\ \mathtt{and\ \ldots\ and\ t}_n\ \mathtt{=}\ \tau_n$$

(where the τ_i may be sum types in which the $\mathtt{t}_i$ may occur). Suppose that some implementation matches this spec under the opaque interpretation; the implementation of each type $\mathtt{t}_i$ must be a recursive type δ_i. Furthermore, the type of the $\mathtt{t}_i\mathtt{\_in}$ function given in the spec is $\tau_i \to \mathtt{t}_i$, and the type of its implementation is $\mathsf{expand}(\delta_i) \to \delta_i$. Because the matching succeeds under the opaque interpretation, we know that these types are equal after each δ_i has been substituted for $\mathtt{t}_i$; thus we know that $\mathsf{expand}(\delta_i) \equiv \tau_i[\vec{\delta}/\vec{\mathtt{t}}]$ for each i.

When the specification is elaborated under the transparent interpretation, however, the resulting signature declares that the implementation of each $\mathtt{t}_i$ is the appropriate projection from a recursive bundle determined by the spec itself. That is, each $\mathtt{t}_i$ is transparently specified as $\mu_i(\vec{\mathtt{t}}).(\vec{\tau})$. In order for the implementation to match this transparent specification, it is thus sufficient to prove the following theorem:

Theorem 1 *If* $\forall i \in 1..n,\ \ \Gamma;\emptyset \vdash \mathsf{expand}(\delta_i) \equiv \tau_i[\vec{\delta}/\vec{\mathtt{t}}]$, *then* $\forall i \in 1..n,\ \ \Gamma;\emptyset \vdash \delta_i \equiv \mu_i(\vec{\mathtt{t}}).(\vec{\tau})$.

Proof: See Appendix A. □

3.5 Discussion

While we have given a formal argument why the trailing version of the boxed-unroll rule is flexible enough to allow the datatype matchings of SML to typecheck under the transparent interpretation, we have not been precise about how maintaining a trail relates to the rest of type equivalence. In fact, the only work regarding trails we are aware of is the seminal work of Amadio and Cardelli [1] on subtyping equi-recursive types, and its later coinductive axiomatization by Brandt and Henglein [2], both of which are conducted in the context of the simply-typed λ-calculus. Our trailing boxed-unroll rule can be viewed as a restriction of the corresponding rule in Amadio and Cardelli's trailing algorithm so that it is only applicable when both types being compared are recursive types.

It is not clear, though, how trails affect more complex type systems that contain type constructors of higher kind, such as Girard's F^ω [6]. In addition to higher kinds, the MIL (Middle Intermediate Language) of TILT employs singleton kinds to model SML's type sharing [13], and the proof that MIL typechecking is decidable is rather delicate and involved. While we have implemented the above trailing algorithm in TILT for experimental purposes (see Section 5), the interaction of trails and singletons is not well-understood.

As for the remaining conflict between the transparent interpretation and type sharing, one might argue that the solution is to broaden SML's semantics for sharing constraints to permit sharing of *rigid* (non-abstract) type components. The problem is that the kind of sharing that would be necessary to make the examples of Section 3.3 typecheck under the transparent interpretation would require some form of type unification. It is difficult to determine where to draw the line between SML's sharing semantics and full higher-order unification, which is undecidable. Moreover, unification would constitute a significant change to SML's semantics, disproportionate to the original problem of efficiently implementing datatypes.

4 A Coercion Interpretation of Datatypes

In this section, we will discuss a treatment of datatypes based on *coercions*. This solution will closely resemble the Harper-Stone interpretation, and thus will not require the boxed-unroll rule or a trail algorithm, but will not incur the run-time cost of a function call at constructor application sites either.

4.1 Representation of Datatype Values

The calculus we have discussed in this paper can be given the usual structured operational semantics, in which an expression of the form $\mathtt{roll}_\delta(v)$ is itself a value if v is a value. (From here on we will assume that the metavariable v ranges only over values.) In fact, it can be shown without difficulty that *any* closed value of a datatype δ must have the form $\mathtt{roll}_\delta(v)$ where v is a closed value of type $\mathsf{expand}(\delta)$. Thus the $\mathtt{roll}$ operator plays a similar role to that of the $\mathtt{inj}_1$ operator for sum types, as far as the high-level language semantics is concerned.

Although we specify the behavior of programs in our language with a formal operational semantics, it is our intent that programs be compiled into machine code for execution, which forces us to take a slightly different view of data. Rather than working directly with high-level language values, compiled programs manipulate *representations* of those values. A compiler is free to choose the representation scheme it uses, provided that the basic operations of the language can be faithfully performed on representations. For example, most compilers construct the value $\mathtt{inj}_1(v)$ by attaching a tag to the value v and storing this new object somewhere. This tagging is necessary in order to implement the $\mathtt{case}$ construct. In particular,

$$
\begin{array}{llcl}
\textit{Types} & \sigma, \tau & ::= & \cdots \mid (\vec{\alpha}; \tau_1) \Rightarrow \tau_2 \\
\textit{Terms} & e & ::= & \cdots \mid \Lambda\vec{\alpha}.\mathtt{fold}_\delta \mid \Lambda\vec{\alpha}.\mathtt{unfold}_\delta \\
 & & & \mid v@(\vec{\tau}; e)
\end{array}
$$

Figure 8: Syntax of Coercions

$$
\frac{\Gamma, \vec{\alpha} \vdash \tau_1 \ \text{type} \quad \Gamma, \vec{\alpha} \vdash \tau_2 \ \text{type}}{\Gamma \vdash (\vec{\alpha}; \tau_1) \Rightarrow \tau_2 \ \text{type}}
\qquad
\frac{\Gamma, \vec{\alpha} \vdash \delta \ \text{type}}{\Gamma \vdash \Lambda\vec{\alpha}.\mathtt{fold}_\delta : (\vec{\alpha}; \mathsf{expand}(\delta)) \Rightarrow \delta}
\qquad
\frac{\Gamma, \vec{\alpha} \vdash \delta \ \text{type}}{\Gamma \vdash \Lambda\vec{\alpha}.\mathtt{unfold}_\delta : (\vec{\alpha}; \delta) \Rightarrow \mathsf{expand}(\delta)}
$$

$$
\frac{\Gamma \vdash v : (\vec{\alpha}; \tau_1) \Rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1[\vec{\sigma}/\vec{\alpha}] \quad \forall i \in 1..n.\ \Gamma \vdash \sigma_i \ \text{type}}{\Gamma \vdash v@(\vec{\sigma}; e) : \tau_2[\vec{\sigma}/\vec{\alpha}]}
$$

Figure 9: Typing Rules for Coercions

the representation of any value of type $\tau_1 + \tau_2$ must carry enough information to determine whether it was created with $\mathtt{inj}_1$ or $\mathtt{inj}_2$ and recover a representation of the injected value.

What are the requirements for representations of values of recursive type? It turns out that they are somewhat weaker than for sums. The elimination form for recursive types is `unroll`, which (unlike `case`) does not need to extract any information from its argument other than the original rolled value. In fact, the only requirement is that a representation of v can be extracted from any representation of $\mathtt{roll}_\delta(v)$. Thus one reasonable representation strategy is to represent $\mathtt{roll}_\delta(v)$ exactly the same as v. In Appendix B, we give a more precise argument as to why this is reasonable, making use of two key insights. First, it is an invariant of the TILT compiler that the representation of any value fits in a single machine register; anything larger than 32 bits is *always* stored in the heap. This means that all possible complications having to do with the sizes of recursive values are avoided. Second, we define representations for values, not types; that is, we define the set of machine words that can represent the value v by structural induction on v, rather than defining the set of words that can represent values of type τ by induction on τ as might be expected.

The TILT compiler adopts this strategy of identifying the representations of $\mathtt{roll}_\delta(v)$ and v, which has the pleasant consequence that the `roll` and `unroll` operations are "no-ops". For instance, the untyped machine code generated by the compiler for the expression $\mathtt{roll}_\delta(e)$ need not differ from the code for e alone, since if the latter evaluates to v then the former evaluates to $\mathtt{roll}_\delta(v)$, and the representations of these two values are the same. The reverse happens for `unroll`.

This, in turn, has an important consequence for datatypes. Since the `in` and `out` functions produced by the HS elaboration of datatypes do nothing but roll or unroll their arguments, the code generated for any `in` or `out` function will be the same as that of the identity function. Hence, the only run-time cost incurred by using an `in` function to construct a datatype value is the overhead of the function call itself. In the remainder of this section we will explain how to eliminate this cost by allowing the types of the `in` and `out` functions to reflect the fact that their implementations are trivial.

4.2 The Coercion Interpretation

To mark `in` and `out` functions as run-time no-ops, we use *coercions*, which are similar to functions, except that they are known to be no-ops and therefore no code needs to be generated for coercion applications. We incorporate coercions into the term level of our language and introduce special coercion types to which they belong. Figure 8 gives the changes to the syntax of our calculus. Note that while we have so far confined our discussion to monomorphic datatypes, the general case of polymorphic datatypes will require polymorphic coercions. The syntax we give here is essentially that used in the TILT compiler; it does not address non-uniform datatypes.

We extend the type level of the language with a type for (possibly polymorphic) coercions, $(\vec{\alpha}; \tau_1) \Rightarrow \tau_2$;

```
structure ExpDec :> sig
     type exp
     type dec
     val exp_in : var + (dec * exp) ⇒ exp
     val exp_out : exp ⇒ var + (dec * exp)
     val dec_in : (var * exp) + (dec * dec) ⇒ dec
     val dec_out : dec ⇒ (var * exp) + (dec * dec)
end = struct
     type exp = μ1(α, β).(var + β * α, var * α + β * β)
     type dec = μ2(α, β).(var + β * α, var * α + β * β)
     val exp_in = fold_exp
     val exp_out = unfold_exp
     val dec_in = fold_dec
     val dec_out = unfold_dec
end
```

Figure 10: Elaboration of exp and dec Under the Coercion Interpretation

$$\textit{Values} \quad v \quad ::= \quad \cdots \mid \Lambda\vec{\alpha}.\texttt{fold}_{\tau} \mid \Lambda\vec{\alpha}.\texttt{unfold}_{\tau} \mid (\Lambda\vec{\alpha}.\texttt{fold}_{\tau})@(\vec{\sigma}; v)$$

$$\frac{e \mapsto e'}{v@(\vec{\tau}; e) \mapsto v@(\vec{\tau}; e')} \qquad \frac{}{(\Lambda\vec{\alpha}.\texttt{unfold}_{\tau_1})@(\vec{\sigma}; ((\Lambda\vec{\beta}.\texttt{fold}_{\tau_2})@(\vec{\sigma}'; v))) \mapsto v}$$

Figure 11: Operational Semantics for Coercions

a value of this type is a coercion that takes $\mathsf{length}(\vec{\alpha})$ type arguments and then can change a value of type τ_1 into one of type τ_2 (where, of course, variables from $\vec{\alpha}$ can appear in either of these types). When $\vec{\alpha}$ is empty, we will write $(\vec{\alpha}; \tau_1) \Rightarrow \tau_2$ as $\tau_1 \Rightarrow \tau_2$.

Similarly, we extend the term level with the (possibly polymorphic) coercion values $\Lambda\vec{\alpha}.\texttt{fold}_{\delta}$ and $\Lambda\vec{\alpha}.\texttt{unfold}_{\delta}$; these take the place of roll and unroll expressions. Coercions are applied to (type and value) arguments in an expression of the form $v@(\vec{\tau}; e)$; here v is the coercion, $\vec{\tau}$ are the type arguments, and e is the value to be coerced. Note that the coercion is syntactically restricted to be a value; this makes the calculus more amenable to a simple code generation strategy, as we will discuss in Section 4.3. The typing rules for coercions are essentially the same as if they were ordinary polymorphic functions, and are shown in Figure 9.

With these modifications to the language in place, we can elaborate the datatypes exp and dec using coercions instead of functions to implement the in and out operations. The result of elaborating this pair of datatypes is shown in Figure 10. Note that the interface is exactly the same as the HS interface shown in Section 2 except that the function arrows (->) have been replaced by coercion arrows (⇒). This interface is implemented by defining exp and dec in the same way as in the HS interpretation, and implementing the in and out coercions as the appropriate fold and unfold values. The elaboration of a constructor application is superficially similar to the opaque interpretation, but a coercion application is generated instead of a function call. For instance, LetExp(d,e) elaborates as $\texttt{exp\_in}@(\texttt{inj}_2(\texttt{d}, \texttt{e}))$.

4.3 Coercion Erasure

We are now ready to formally justify our claim that coercions may be implemented by *erasure*, that is, that it is sound for a compiler to consider coercions only as "retyping operators" and ignore them when generating code. First, we will describe the operational semantics of the coercion constructs we have added to our internal language. Next, we will give a translation from our calculus into an untyped one in which coercion applications disappear. Finally, we will state a theorem guaranteeing that the translation is safe.

$$M \quad ::= \cdots \mid \lambda x.M \mid \texttt{fold} \mid \texttt{unfold}$$

$$\begin{aligned} x^- &= x \\ (\lambda x{:}\tau.e)^- &= \lambda x.e^- \\ (\Lambda\vec{\alpha}.\texttt{fold}_\delta)^- &= \texttt{fold} \\ (\Lambda\vec{\alpha}.\texttt{unfold}_\delta)^- &= \texttt{unfold} \\ (v@(\vec{\tau};e))^- &= e^- \\ &\vdots \end{aligned}$$

Figure 12: Target Language Syntax; Type and Coercion Erasure

The operational semantics of our coercion constructs are shown in Figure 11. We extend the class of values with the `fold` and `unfold` coercions, as well as the application of a `fold` coercion to a value. These are the canonical forms of coercion types and recursive types respectively. The two inference rules shown in Figure 11 define the manner in which coercion applications are evaluated. The evaluation of a coercion application is similar to the evaluation of a normal function application where the applicand is already a value. The rule on the left specifies that the argument is reduced until it is a value. If the applicand is a `fold`, then the application itself is a value. If the applicand is an `unfold`, then the argument must have a recursive type and therefore (by canonical forms) consist of a `fold` applied to a value v. The rule on the right defines `unfold` to be the left inverse of `fold`, and hence this evaluates to v.

As we have already discussed, the data representation strategy of TILT is such that no code needs to be generated to compute `fold` v from v, nor to compute the result of cancelling a `fold` with an `unfold`. Thus it seems intuitive that to generate code for a coercion application $v@(\vec{\tau};e)$, the compiler can simply generate code for e, with the result that datatype constructors and destructors under the coercion interpretation have the same run-time costs as Harper and Stone's functions would if they were inlined. To make this more precise, we now define an *erasure* mapping to translate terms of our typed internal language into an untyped language with no coercion application. The untyped nature of the target language (and of machine language) is important: treating v as `fold` v would destroy the subject reduction property of a typed language.

Figure 12 gives the syntax of our untyped target language and the coercion-erasing translation. The target language is intended to be essentially the same as our typed internal language, except that all types and coercion applications have been removed. It contains untyped coercion values `fold` and `unfold`, but no coercion application form. The erasure translation turns expressions with type annotations into expressions without them (λ-abstraction and coercion values are shown in the figure), and removes coercion applications so that the erasure of $v@(\vec{\tau};e)$ is just the erasure of e. In particular, for any value v, v and `fold` v are identified by the translation, which is consistent with our intuition about the compiler. The operational semantics of the target language is analogous to that of the source.

The language with coercions has the important type-safety property that if a term is well-typed, its evaluation does not get stuck. An important theorem is that the coercion-erasing translation preserves the safety of well-typed programs:

Theorem 2 (Erasure Preserves Safety) *If $\Gamma \vdash e : \tau$, then e^- is safe. That is, if $e^- \mapsto^* f$, then f is not stuck.*

Proof: See Appendix C. □

Note that the value restriction on coercions is crucial to the soundness of this "coercion erasure" interpretation. Since a divergent expression can be given an arbitrary type, including a coercion type, any semantics in which a coercion expression is not evaluated before it is applied fails to be type-safe. Thus if arbitrary expressions of coercion type could appear in application positions, the compiler would have to generate code for them. Since values cannot diverge or have effects, we are free to ignore coercion applications when we generate code.

Test	HS	CID	TID
life	8.233	2.161	2.380
leroy	5.497	4.069	3.986
fft	22.167	17.619	16.509
boyer	2.031	1.559	1.364
simple	1.506	1.003	0.908
tyan	16.239	8.477	9.512
msort	1.685	0.860	1.012
pia	1.758	1.494	1.417
lexgen	11.052	5.599	5.239
frank	37.449	25.355	26.017
TOTAL	107.617	68.199	68.344

Figure 13: Performance Comparison

5 Performance

To evaluate the relative performance of the different interpretations of datatypes we have discussed, we performed experiments using three different versions of the TILT compiler: one that implements a naïve Harper-Stone interpretation in which the construction of a non-locally-defined datatype requires a function call[2]; one that implements the coercion interpretation of datatypes; and one that implements the transparent interpretation. We compiled ten different benchmarks using each version of the compiler; the running times for the resulting executables (averaged over three trials) are shown in Figure 13. All tests were run on an Ultra-SPARC Enterprise server; the times reported are CPU time in seconds.

The measurements clearly indicate that the overhead due to datatype constructor function calls under the naïve HS interpretation is significant. The optimizations afforded by the coercion and transparent interpretations provide comparable speedups over the opaque interpretation, both on the order of 37% (comparing the total running times). Given that, of the two optimized approaches, only the coercion interpretation is entirely faithful to the semantics of SML, and since the theory of coercion types is a simpler and more orthogonal extension to the HS type theory than the trailing algorithm of Section 3.4, we believe the coercion interpretation is the more robust choice.

6 Related Work

Our trail algorithm for weakened recursive type equivalence is based on the one implemented by Shao in the FLINT intermediate language of the Standard ML of New Jersey compiler [12]. The typing rules in Section 3.4 are based on the formal semantics for FLINT given by League and Shao [8], although we are the first to give a formal argument that their trailing algorithm actually works. It is important to note that SML/NJ only implements the transparent interpretation *internally*: the opaque interpretation is employed during elaboration, and datatype specifications are made transparent only afterward. As the examples of Section 3.3 illustrate, there are programs that typecheck according to SML but not under the transparent interpretation even with trailing equivalence, so it is unclear what SML/NJ does (after elaboration) in these cases. As it happens, the final example of Section 3.3, which is valid SML, is rejected by the SML/NJ compiler.

Curien and Ghelli [4] and Crary [3] have defined languages that use coercions to replace subsumption rules in languages with subtyping. Crary's calculus of coercions includes `roll` and `unroll` for recursive types, but since the focus of his paper is on subtyping he does not explore the potential uses of these coercions in detail. Nevertheless, our notion of coercion erasure, and the proof of our safety preservation theorem, are based on Crary's. The implementation of Typed Assembly Language for the x86 architecture (TALx86) [10] allows operands to be annotated with coercions that change their types but not their representations; these coercions include `roll` and `unroll` as well as introduction of sums and elimination of universal quantifiers.

[2] In particular, we implement the strategy described at the end of Section 2.2.

Our intermediate language differs from these in that we include coercions in the term level of the language rather than treating them specially in the syntax. This simplifies the presentation of the coercion interpretation of datatypes, and it simplified our implementation because it required a smaller incremental change from earlier versions of the TILT compiler. However, including coercions in the term level is a bit unnatural, and our planned extension of TILT with a type-preserving back-end will likely involve a full coercion calculus.

7 Conclusion

The generative nature of SML datatypes poses a significant challenge for efficient type-preserving compilation. Generativity can be correctly understood by interpreting datatypes as structures that hold their type components abstract, exporting functions that construct and deconstruct datatype values. Under this interpretation, the inlining of datatype construction and deconstruction operations is not type-preserving and hence cannot be performed by a typed compiler such as TILT.

In this paper, we have discussed two approaches to eliminating the function call overhead in a type-preserving way. The first, doing away with generativity by making the type components of datatype structures transparent, results in a new language that is different from, but neither more nor less permissive than, Standard ML. Some of the lost expressiveness can be regained by relaxing the rules of type equivalence in the intermediate language, at the expense of complicating the type theory. The fact that the transparent interpretation forbids datatypes to appear in `sharing type` or `where type` signature constraints is unfortunate; it is possible that a revision of the semantics of these constructs could remove the restriction.

The second approach, replacing the construction and deconstruction functions of datatypes with coercions that may be erased during code generation, eliminates the function call overhead without changing the static semantics of the external language. However, the erasure of coercions only makes sense in a setting where a recursive-type value and its unrolling are represented the same at run time. The coercion interpretation of datatypes has been implemented in the TILT compiler.

Although we have presented our analysis of SML datatypes in the context of Harper-Stone and the TILT compiler, the idea of "coercion types" is one that we think is generally useful. Terms that serve only as retyping operations are pervasive in typed intermediate languages, and are usually described as "coercions" that can be eliminated before running the code. However, applications of these informal coercions cannot in general be erased if there is no way to distinguish coercions from ordinary functions by their types; this is a problem especially in the presence of true separate compilation. Our contribution is to provide a simple mechanism that permits coercive terms to be recognized as such and their applications to be safely eliminated, without requiring significant syntactic and meta-theoretic overhead.

References

[1] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[2] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998. Invited submission to special issue featuring a selection of contributions to the 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), 1997.

[3] Karl Crary. Typed compilation of inclusive subtyping. In *2000 ACM International Conference on Functional Programming*, Montreal, September 2000.

[4] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[5] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *2000 ACM International Conference on Functional Programming*, 2000. To appear in *Journal of Functional Programming.*

[6] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[7] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Robin Milner Festschrifft.* MIT Press, 1998.

[8] Christopher League and Zhong Shao. Formal semantics of the FLINT intermediate language. Technical Report Yale-CS-TR-1171, Yale University, 1998.

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[10] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[11] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, School of Computer Science, Carnegie Mellon University, December 2000.

[12] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[13] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 214–227, Boston, January 2000.

[14] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

A Proof of Theorem 1

Suppose that $\forall i \in 1..n$, $\Gamma;\emptyset \vdash \mathsf{expand}(\delta_i) \equiv \tau_i[\vec{\delta}/\vec{\mathtt{t}}]$. Then we can prove the following lemma:

Lemma 1 *For any set $S \subseteq \{1,\ldots,n\}$, define $A_S = \{\delta_i \equiv \mu_i(\vec{\mathtt{t}}).(\vec{\tau}) \mid i \in S\}$. Then for any $S \subseteq \{1,\ldots,n\}$ and any $j \in \{1,\ldots,n\}$, $\Gamma; A_S \vdash \delta_j \equiv \mu_j(\vec{\mathtt{t}}).(\vec{\tau})$.*

Proof Sketch: The proof is by induction on $n - |S|$. If $n - |S| = 0$, then for any j the required equivalence is an assumption in A_S and can therefore be concluded using the assumption rule. If $n - |S| > 0$, then there are two cases:

> **Case:** $j \in S$. Then the required equivalence is an assumption in A_S.
>
> **Case:** $j \notin S$. Then let $S' = S \cup \{j\}$. Note that $|S'| > |S|$ and so $n - |S'| < n - |S|$. By the induction hypothesis, $\Gamma; A_{S'} \vdash \delta_k \equiv \mu_k(\vec{\mathtt{t}}).(\vec{\tau})$ for every $k \in \{1,\ldots,n\}$. Because substituting equal types into equal types gives equal results, $\Gamma; A_{S'} \vdash \tau_j[\vec{\delta}/\vec{\mathtt{t}}] \equiv \tau_j[\vec{\mu}(\vec{\mathtt{t}}).(\vec{\tau})/\vec{\mathtt{t}}]$. By assumption, $\mathsf{expand}(\delta_j) \equiv \tau_j[\vec{\delta}/\vec{\mathtt{t}}]$, so by transitivity $\Gamma; A_{S'} \vdash \mathsf{expand}(\delta_j) \equiv \tau_j[\vec{\mu}(\vec{\mathtt{t}}).(\vec{\tau})/\vec{\mathtt{t}}]$. The type on the right side of this equivalence is just $\mathsf{expand}(\mu_j(\vec{\mathtt{t}}).(\vec{\tau}))$, so by the trailing boxed-unroll rule we can conclude $\Gamma; A_S \vdash \delta_j \equiv \mu_j(\vec{\mathtt{t}}).(\vec{\tau})$, as required. □

The desired result then follows as a corollary:

Corollary 1 *For $j \in \{1,\ldots,n\}$, $\Gamma;\emptyset \vdash \delta_j \equiv \mu_j(\vec{\mathtt{t}}).(\vec{\tau})$.*

Proof: Choose $S = \emptyset$. By the Lemma, $\Gamma; A_\emptyset \vdash \delta_j \equiv \mu_j(\vec{\mathtt{t}}).(\vec{\tau})$. But $A_\emptyset = \emptyset$, so we are done. □

$$\begin{array}{lccl} \textit{Types} & \sigma,\tau & ::= & \mathtt{nat} \mid \langle\tau_1,\ldots,\tau_k\rangle \mid [\tau_1,\ldots,\tau_k] \mid \mu\alpha.\tau \\ \textit{Values} & v & ::= & \overline{n} \mid (v_0,\ldots,v_{m-1}) \mid \mathtt{inj}_i\, v \mid \mathtt{roll}_{\mu\alpha.\tau}(v) \end{array}$$

Figure 14: Syntax of Types and Values in the First-Order Fragment of a Specialized Intermediate Language

$$\begin{array}{lll} H \vdash n \lhd \overline{n} : \mathtt{nat} & & \\ H \vdash n \lhd (v_0,\ldots,v_{m-1}) : \langle\tau_0,\ldots,\tau_{m-1}\rangle & \text{if} & H \vdash H(n+i) \lhd v_i : \tau_i \text{ for each } i,\ 0 \le i < m \\ H \vdash n \lhd \mathtt{inj}_i\, v : [\tau_1,\ldots,\tau_k] & \text{if} & 1 \le i < k \text{ and } H(n) = i \text{ and} \\ & & H \vdash H(n+1) \lhd v : \tau_i \\ H \vdash n \lhd \mathtt{roll}_{\mu\alpha.\tau}\, v : \mu\alpha.\tau & \text{if} & H \vdash n \lhd v : \tau[\mu\alpha.\tau/\alpha]. \end{array}$$

Figure 15: Representation Strategy for the Intermediate Language Fragment

B Data Representation

In this appendix we will give a formal description of a data representation strategy similar to the one used by the TILT compiler. This is intended to clarify the sense in which the `roll` and `unroll` operations on recursive types are "no-ops" at run time.

A key invariant of the TILT system is that every value (except floating-point numbers) manipulated by a program at run-time is 32 bits wide. All values of record and sum types that require more than 32 bits of storage are *boxed*—*i.e.*, stored in the heap and referred to by a 32-bit pointer value. Our formalization of data representation will therefore attempt to characterize when, in the context of a particular heap, a particular integer represents a particular source-language value. Once we have done this, we will argue that the representation strategy we have defined is *reasonable*—that is, that all the operations a program must perform on values may be faithfully performed on their representations under this strategy.

To begin, we make the simplifying assumption that any integer may be a valid memory address, and that a memory location can hold any integer. This can easily be restricted so that only integers between 0 and, say, 2^{32} are used, but allowing arbitrary integers makes the presentation easier. Under these assumptions about the world, it makes sense to define heaps as follows:

Definition 1 *A* heap *is a finite partial function* $H : \mathrm{N} \to \mathrm{N}$.

Next, we define the general notion of a representation strategy.

Definition 2 *A* representation strategy *is a four-place relation* S *on heaps, natural numbers, closed values and closed types such that if* $(H, n, v, \tau) \in S$ *and* $H \subseteq H'$*, then* $(H', n, v, \tau) \in S$.

If S is a representation strategy, we will use the notation $H \vdash_S n \lhd v : \tau$ to mean that $(H, n, v, \tau) \in S$, omitting the subscript S if it is clear from context. That statement may be read as "in the heap H, the number n represents the value v at type τ."

We will now proceed to define a particular representation strategy, similar to the one used by the TILT compiler. Figure 14 gives the syntax of the types and terms we will be representing. The types include the type of integers, k-ary tuple types $\langle\tau_1,\ldots,\tau_k\rangle$, k-ary sum types $[\tau_1,\ldots,\tau_k]$ and (single) recursive types $\mu\alpha.\tau$. (We are not considering arrow types or function values here, because they complicate the presentation in ways not relevant to recursive types.)

Figure 15 gives the definition of one possible representation strategy for these values. Note that this strategy is well-defined, because all the values on the right-hand side of each clause are syntactically smaller than the one on the left. An integer value $\overline{n}$ is represented by the integer n. A tuple of k values is represented by a pointer (n) giving the location of the first of k consecutive heap cells containing representations of the tuple's components. An injection into a sum type is represented by a pointer to what is essentially a two-element tuple: the first element is a tag that identifies a branch of the sum type, and the second is a representation of the injected value. Finally, a value of recursive type is represented by a representation of the rolled value.

This data representation strategy is similar to, but considerably simpler than, the one used by the TILT/ML compiler. The main difference is in the handling of sum types; TILT uses a more complex version of sum type representations that saves space and improves run-time efficiency. Fortunately, the treatment of sums and the treatment of recursive types are orthogonal, so the differences between the representation strategy in Figure 15 and that of TILT is unimportant.

We can now argue that this data representation strategy is reasonable. The following property may be proved using the definition.

Construction Property: *Let H be a heap. If $\emptyset \vdash v : \tau$, then there is a heap $H' \supseteq H$ and an integer n such that $H' \vdash n \lhd v : \tau$.*

The construction property states that any well-typed closed value can be represented in some extension of any initial heap. This roughly means that all the introduction forms of the language can be faithfully implemented under our representation strategy. It remains to argue that the elimination forms may be implemented as well.

The elimination forms for integers are arithmetic operations and comparisons. Since the representations of integer values are the integers themselves, these operations can be implemented. The elimination form for tuples is projection; we need to show that if $1 \leq i \leq k$ and $H \vdash n \lhd v : \langle \tau_0, \ldots, \tau_{k-1} \rangle$ then we can find some n' that represents the value of $\pi_i\, v$. By canonical forms, the value v must be $(v_0, \ldots, v_{k-1})$, and so $\pi_i\, v$ evaluates to v_i. But by the definition of our representation strategy, $H \vdash H(n+i) \lhd v_i : \tau_i$, so projection can be implemented.

The elimination form for sums, the `case` construct, is a little different in that it may take many more than one step to produce a value, if it produces one at all. Only the first of these steps, however—the one in which the `case` chooses one of its subexpressions to continue executing and passes the appropriate value to that branch—is relevant to the reasonableness of our representation for sums. In particular, if $\emptyset \vdash v : (\tau_1, \ldots, \tau_k)$ then the expression

$$\texttt{case}\, v\, \texttt{of}\, \texttt{inj}_1\, x \Rightarrow e_1 \mid \ldots \mid \texttt{inj}_k\, x \Rightarrow e_k$$

will certainly take a step to $e_j[v'/x]$ for some j and some value v' of the appropriate type. In order to show `case` is implementable, it suffices to show that given $H \vdash n \lhd v : [\tau_1, \ldots, \tau_k]$ we can compute the appropriate j and a representation of the appropriate value v'. But note that by canonical forms v must have the form $\texttt{inj}_i\, v_i$, in which case the branch to select is the i^{th} one, and the value to pass is v_i (*i.e.*, $j = i$ and $v' = v_i$). According to our representation strategy, $H(n) = i$ and $H \vdash H(n+1) \lhd v_i : \tau_i$, so we can implement `case`.

Finally, consider the elimination form for recursive types, $\texttt{unroll}_{\mu\alpha.\tau}$. In order to implement this operation, it must be the case that if $\emptyset \vdash v : \mu\alpha.\tau$ and $H \vdash n \lhd v : \mu\alpha.\tau$ then we can construct some n' such that n' represents the value of $\texttt{unroll}_{\mu\alpha.\tau}\, v$. By canonical forms, $v = \texttt{roll}_{\mu\alpha.\tau}\, v'$ and $\texttt{unroll}_{\mu\alpha.\tau}\, v$ evaluates to v'. But under our representation strategy, $H \vdash n \lhd v' : \tau[\mu\alpha.\tau/\alpha]$ and so we can implement `unroll`.

Notice also that a representation of a value v of recursive type also represents the value produced by $\texttt{unroll}\, v$, just as was the case for `roll`. This justifies our notion that `roll` and `unroll` may be implemented as no-ops in a compiler that uses this data representation strategy.

C Proof of Coercion Erasure

In this appendix we will prove the coercion erasure safety theorem stated in Section 4 for a simple fragment of our intermediate language. The theorem states that if an expression is well-typed, then evaluation of its erasure does not get stuck; that is, it never reaches an expression that is not a value but for which no transition rule applies. To prove this, we will establish a correspondence between the transition relation of the typed calculus and that of the untyped one; we will then be able to show that the type-preservation and progress lemmas for the typed calculus guarantee safety of their erasures. The proofs of type preservation and progress for the typed language are standard and omitted.

$$\frac{}{(\lambda x{:}\tau\,.\,e_1)\,v_2 \mapsto_{\mathtt{e}} e_1[v_2/x]} \qquad \frac{}{(\Lambda\vec{\alpha}.\mathtt{unfold}_{\tau_1})@(\vec{\sigma};((\Lambda\vec{\beta}.\mathtt{fold}_{\tau_2})@(\vec{\sigma}';v))) \mapsto_{\mathtt{c}} v}$$

$$\frac{e_1 \mapsto_{\varphi} e_1'}{e_1\,e_2 \mapsto_{\varphi} e_1'\,e_2} \qquad \frac{e_2 \mapsto_{\varphi} e_2'}{v_1\,e_2 \mapsto_{\varphi} v_1\,e_2'} \qquad \frac{e \mapsto_{\varphi} e'}{v@(\vec{\tau};e) \mapsto_{\varphi} v@(\vec{\tau};e')}$$

Figure 16: Annotated Operational Semantics for Typed Language

The first two lemmas we will use are easy to prove using the definition of erasure given in Section 4.

Lemma 2 (Erasure of Values) *If v is a value, then v^- is a value.*

Proof: By structural induction on v. □

Lemma 3 (Substitution and Erasure Commute)
For any expression e and value v, $(e[v/x])^- = e^-[v^-/x]$.

Proof: By structural induction on e. □

In order to properly characterize the relationship between the typed and untyped transition relations we must distinguish, for the typed calculus, between evaluation steps like β-reduction that correspond to steps in the untyped semantics and those like coercion application that do not. To this end, we annotate transitions of the typed calculus as shown in Figure 16. The flag φ adorning each transition may be either $\mathtt{e}$, indicating an "evaluation" step that is preserved by erasure, or $\mathtt{c}$, indicating (to use Crary's [3] terminology) a "canonicalization" step that is removed by erasure. We will continue to use the unannotated $\mapsto$ to stand for the union of $\mapsto_{\mathtt{e}}$ and $\mapsto_{\mathtt{c}}$, and as usual we will use $\mapsto^*$, $\mapsto_{\mathtt{e}}^*$ and $\mapsto_{\mathtt{c}}^*$ to denote the reflexive, transitive closures of these relations. With these definitions in place, we can prove the following lemma, which states that evaluation steps are preserved by erasure, but terms that differ by a canonicalization step are identified by erasure. It follows that any sequence of steps in the typed language erases to some sequence of steps in the untyped language.

Lemma 4 (Simulation)

1. *If $e_1 \mapsto_{\mathtt{e}} e_2$, then $e_1^- \mapsto e_2^-$.*
2. *If $e_1 \mapsto_{\mathtt{c}} e_2$, then $e_1^- = e_2^-$.*

Proof: By induction on derivations, using the definition of erasure and Lemmas 2 and 3. □

Lemma 5

1. *If $e_1 \mapsto_{\mathtt{c}}^* e_2$, then $e_1^- = e_2^-$.*
2. *If $e_1 \mapsto_{\mathtt{e}}^* e_2$, then $e_1^- \mapsto^* e_2^-$.*
3. *If $e_1 \mapsto^* e_2$, then $e_1^- \mapsto^* e_2^-$.*

Proof: Each part is proved separately by induction on the length of the transition sequence. □

We are now ready to prove the equivalent of a Progress lemma for the untyped language. It states that a term whose erasure is a value canonicalizes to a value in some number of steps, and a term whose erasure is not a value will eventually take an evaluation step.

Lemma 6 (Progress under Erasure) *If $\emptyset \vdash e : \tau$ then either*

1. *e^- is a value and $e \mapsto_{\mathtt{c}}^* v$ for some value v, OR*

2. e^- is not a value and $e \mapsto_{\mathsf{c}}^* e' \mapsto_{\mathsf{e}} e''$ for some e' and e''.

Proof: By induction on the typing derivation for e. Note that if e is a value, then so is e^- (by Lemma 2) and $e \mapsto_{\mathsf{c}}^* e$ by definition. Thus we need only consider the typing rules in which the expression being typed may be a non-value.

Case:

$$\frac{\emptyset \vdash e_1 : \tau' \to \tau \quad \emptyset \vdash e_2 : \tau'}{\emptyset \vdash e_1\, e_2 : \tau}$$

Note that $(e_1\, e_2)^- = e_1^-\, e_2^-$, which is not a value. Thus we must show that $e_1\, e_2 \mapsto_{\mathsf{c}}^* e' \mapsto_{\mathsf{e}} e''$. There are three sub-cases:

Sub-case: e_1^- is not a value. By the induction hypothesis, $e_1 \mapsto_{\mathsf{c}}^* e_1' \mapsto_{\mathsf{e}} e_1''$. Thus, $e_1\, e_2 \mapsto_{\mathsf{c}}^* e_1'\, e_2 \mapsto_{\mathsf{e}} e_1''\, e_2$.

Sub-case: e_1^- is a value, e_2^- is not. By the induction hypothesis, $e_1 \mapsto_{\mathsf{c}}^* v_1$ and $e_2 \mapsto_{\mathsf{c}}^* e_2' \mapsto_{\mathsf{e}} e_2''$. Thus, $e_1\, e_2 \mapsto_{\mathsf{c}}^* v_1\, e_2 \mapsto_{\mathsf{c}}^* v_1\, e_2' \mapsto_{\mathsf{e}} v_1\, e_2''$.

Sub-case: e_1^- and e_2^- are both values. By the induction hypothesis, $e_1 \mapsto_{\mathsf{c}}^* v_1$ and $e_2 \mapsto_{\mathsf{c}}^* v_2$. By type preservation, $\emptyset \vdash v_1 : \tau' \to \tau$. By canonical forms, $v_1 = \lambda x{:}\tau'.e_3$. Thus, $e_1\, e_2 \mapsto_{\mathsf{c}}^* (\lambda x{:}\tau'.e_3)\, e_2 \mapsto_{\mathsf{c}}^* (\lambda x{:}\tau'.e_3)\, v_2 \mapsto_{\mathsf{e}} e_3[v_2/x]$.

Case:

$$\frac{\begin{array}{c}\Gamma \vdash v_c : (\vec{\alpha}; \tau_1) \Rightarrow \tau_2 \\ \Gamma \vdash e_1 : \tau_1[\vec{\sigma}/\vec{\alpha}] \quad \forall i \in 1..n.\ \Gamma \vdash \sigma_i \text{ type}\end{array}}{\Gamma \vdash v_c@(\vec{\sigma}; e_1) : \tau_2[\vec{\sigma}/\vec{\alpha}]}$$

Note that $(v_c@(\vec{\sigma}; e_1))^- = e_1{}^-$. If $e_1{}^-$ is not a value, then by the induction hypothesis we get $e_1 \mapsto_{\mathsf{c}}^* e_1' \mapsto_{\mathsf{e}} e_1''$, and hence $v_c@(\vec{\sigma}; e_1) \mapsto_{\mathsf{c}}^* v_c@(\vec{\sigma}; e_1') \mapsto_{\mathsf{e}} v_c@(\vec{\sigma}; e_1'')$ as required.

If $e_1{}^-$ is a value, then we must show that $v_c@(\vec{\sigma}; e_1) \mapsto_{\mathsf{c}}^* v$ for some value v. By the induction hypothesis we have $e_1 \mapsto_{\mathsf{c}}^* v_1$. There are two sub-cases.

Sub-case: The coercion v_c is $\Lambda\vec{\alpha}.\mathtt{fold}_\delta$. Then $v_c@(\vec{\sigma}; v_1)$ is a value and $v_c@(\vec{\sigma}; e_1) \mapsto_{\mathsf{c}}^* v_c@(\vec{\sigma}; v_1)$ as required.

Sub-case: The coercion v_c is $\Lambda\vec{\alpha}.\mathtt{unfold}_\delta$. By inversion, $\tau_1 \equiv \delta$. By type preservation, $\emptyset \vdash v_1 : \delta[\vec{\sigma}/\vec{\alpha}]$. By canonical forms, $v_1 = (\Lambda\vec{\beta}.\mathtt{fold}_{\delta'})@(\vec{\sigma}'; v_1')$ for some δ' and $\vec{\sigma}'$. Thus, $v_c@(\vec{\sigma}; e_1) \mapsto_{\mathsf{c}}^* v_c@(\vec{\sigma}; v_1) = (\Lambda\vec{\alpha}.\mathtt{unfold}_\delta)@(\vec{\sigma}; (\Lambda\vec{\beta}.\mathtt{fold}_{\delta'})@(\vec{\sigma}'; v_1')) \mapsto_{\mathsf{c}} v_1'$. □

Next, we would like to prove an analogue of type preservation for our target calculus. Clearly it is meaningless to prove type preservation for an untyped calculus, so we must prove instead that if the erasure of a well-typed term takes a step, the result is itself the erasure of a well-typed term. Because type preservation does hold for the typed calculus, the following lemma is sufficient to show this.

Lemma 7 (Preservation under Erasure) *If $\emptyset \vdash e : \tau$ and $e^- \mapsto f$, then there is some e' such that $e \mapsto^* e'$ and $(e')^- = f$.*

Proof: By induction on the typing derivation for e. Note that since $e^- \mapsto f$, e^- cannot be a value and hence neither can e. Thus, as for the progress lemma, we only need to consider typing rules in which the expression being typed may be a non-value.

Case:

$$\frac{\begin{array}{c}\emptyset \vdash v_c : (\vec{\alpha}; \tau_1) \Rightarrow \tau_2 \\ \emptyset \vdash e_1 : \tau_1[\vec{\sigma}/\vec{\alpha}] \quad \forall i \in 1..n.\ \emptyset \vdash \sigma_i \text{ type}\end{array}}{\emptyset \vdash v_c@(\vec{\sigma}; e_1) : \tau_2[\vec{\sigma}/\vec{\alpha}]}$$

Note that $(v_c@(\vec{\sigma}; e_1))^- = {e_1}^-$, so in fact ${e_1}^- \mapsto f$. By the induction hypothesis, $e_1 \mapsto^* e_1'$ where $(e_1')^- = f$. Thus $v_c@(\vec{\sigma}; e_1) \mapsto^* v_c@(\vec{\sigma}; e_1')$, and $(v_c@(\vec{\sigma}; e_1'))^- = (e_1')^- = f$.

Case:

$$\frac{\emptyset \vdash e_1 : \tau' \rightarrow \tau \quad \emptyset \vdash e_2 : \tau'}{\emptyset \vdash e_1\, e_2 : \tau}$$

Note that $(e_1\, e_2)^- = {e_1}^-\, {e_2}^-$, and so ${e_1}^-\, {e_2}^- \mapsto f$. There are three possibilities for the last rule used in the derivation of this transition.

Sub-case:

$$\frac{{e_1}^- \mapsto f_1}{{e_1}^-\, {e_2}^- \mapsto f_1\, {e_2}^-} \qquad \text{(where } f = f_1\, {e_2}^-\text{)}$$

By the induction hypothesis, $e_1 \mapsto^* e_1'$ such that ${e_1'}^- = f_1$. Thus $e_1\, e_2 \mapsto^* e_1'\, e_2$ and $(e_1'\, e_2)^- = (e_1')^-\, {e_2}^- = f_1\, {e_2}^- = f$.

Sub-case:

$$\frac{{e_2}^- \mapsto f_2}{w_1\, {e_2}^- \mapsto w_1\, f_2}$$

(where ${e_1}^- = w_1$ is a value, and $f = w_1\, f_2$)

By Lemma 6, $e_1 \mapsto^* v_1$. By Lemma 5, ${v_1}^- = {e_1}^-$; hence ${v_1}^- = w_1$. By the induction hypothesis, $e_2 \mapsto^* e_2'$ such that $(e_2')^- = f_2$. Thus, $e_1\, e_2 \mapsto^* v_1\, e_2 \mapsto^* v_1\, e_2'$, and $(v_1\, e_2')^- = {v_1}^-\, (e_2')^- = w_1\, f_2 = f$.

Sub-case:

$$\frac{}{(\lambda x.f_3)\, w_2 \mapsto f_3[w_2/x]}$$

(where ${e_1}^- = \lambda x.f_3$, ${e_2}^- = w_2$ is a value,
and $f = f_3[w_2/x]$.)

By Lemma 6, $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$. By Lemma 5, ${v_1}^- = \lambda x.f_3$ and ${v_2}^- = w_2$. By type preservation, $\emptyset \vdash v_1 : \tau' \rightarrow \tau$, and so by canonical forms $v_1 = \lambda x : \tau'.e_3$. But this means that ${v_1}^- = \lambda x.{e_3}^-$, so it must be the case that ${e_3}^- = f_3$. Thus we have

$$\begin{aligned} e_1\, e_2 &\mapsto^* (\lambda x{:}\tau'\ .\ e_3)\, e_2 \\ &\mapsto^* (\lambda x{:}\tau'\ .\ e_3)\, v_2 \mapsto^* e_3[v_2/x] \end{aligned}$$

and by Lemma 3, $(e_3[v_2/x])^- = {e_3}^-[{v_2}^-/x] = f_3[w_2/x] = f$. □

We can now extend this lemma to transition sequences of any length, and prove the safety theorem. This last theorem effectively states that the erasure of a well-typed expression can never get stuck.

Lemma 8 *If $\emptyset \vdash e : \tau$ and $e^- \mapsto^* f$, then there is some e' such that $e \mapsto^* e'$ and $(e')^- = f$.*

Proof: By induction on the length of the transition sequence. □

Theorem 3 (Erasure Preserves Safety) *If $\emptyset \vdash e : \tau$ and $e^- \mapsto^* f$, then either f is a value or there exists an f' such that $f \mapsto f'$.*

Proof: By Lemma 8, there is an e' such that $e \mapsto^* e'$ and $(e')^- = f$. By type preservation, $\emptyset \vdash e' : \tau$. Suppose f is not a value; then by Lemma 6 there are e'' and e''' such that $e' \mapsto_{\mathsf{c}}^* e'' \mapsto_{\mathsf{e}} e'''$. By Lemma 5, $(e')^- = (e'')^-$ and $(e'')^- \mapsto (e''')^-$. Therefore, $f \mapsto (e''')^-$. □