The Theory and Implementation of Resource Aware ML 2

Ethan Chu

CMU-CS-23-140

December 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jan Hoffmann, *Chair*
Frank Pfenning

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Keywords: Amortized Analysis, Resource Analysis, Type System, Linear Programming

For my dog William

Abstract

Resource Aware ML (RaML) is a tool that can compute the resource use of programs in the ML family (SML, OCaml, etc) through a type-based technique known as automatic amortized resource analysis (AARA). Existing implementations of RaML have numerous shortcomings, such as lacking support for regular recursive types and being difficult to maintain.

This thesis presents the theory and implementation of a new and improved version of RaML, Resource Aware ML 2. Using work by Grosen et al, RaML 2 can analyze programs with regular recursive types, non-monotone resource analysis, and more. Furthermore, it presents the univariate AARA theory that the implementation is based on; univariate AARA's expressive power is a subset of Grosen et al's multivariate AARA, but it is still a unique variant.

Additionally, this thesis presents supporting features for a full RaML implementation. This includes a new Intermediate Representation (IR) that the resource analysis operates on; this IR is implemented efficiently with an infrastructure known as Abstract Binding Trees. Other contributions include an SML frontend implementation that can translate arbitrary SML code to the RaML IR, as well as a linear programming solver backend.

Acknowledgments

Foremost, I want to thank my thesis committee, Professors Jan Hoffmann and Frank Pfenning. As the professor of my Foundation of Programming Languages course 15-312, Jan is responsible for bringing me into the beautiful world of PL. His encouragement brought me back to CMU as a master's student, steering me toward the righteous path of PL research. This thesis would not exist without his guidance. I would like to thank Frank too, for his detailed and insightful feedback on this thesis. Furthermore, I am grateful for his course on Substructural Logics, which solidified my love for PL and logic.

I also want to thank my peers in PL. Jessie Grosen and David Kahn not only authored works that heavily impact this thesis, but have also sat down with me and provided invaluable advice, both technical and personal, during the course of my master's. I thank Harrison Grodin too; his stellar TA skills in 312 and other PL adjacent courses definitely helped pull me into the field, and we also had fruitful discussions about my thesis. As for Bretton Chen, I am proud to call him my friend, and I am eternally grateful that he planted the seed of PL in my mind back in high school.

Last but not least, I want to thank my parents for their unwavering support of me throughout my life and academic journey at CMU.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Automatic Amortized Resource Analysis

Automatic Amortized Resource Analysis (AARA), first introduced by Hoffmann et al [5], is a type-based technique for automatically inferring symbolic resource bounds at compile time. Unlike its peers which generate bounds that are functions of input list size or other simple metrics, AARA is capable of deriving bounds for complex data structures like lists of lists. Using the physicist's method of amortized analysis developed by Tarjan [10], it can assign potential functions to complex data structures and track the potential available at any point in the program.

AARA's type system consists of local inference rules — a derivation tree using these inference rules is a proof certifying resource bounds for a program. AARA can also do type inference, generating resource bounds in the form of linear inequalities. The potential functions used in the physicist's method are built from carefully selected base polynomials, and the type system generates constraints solely between their coefficients. This innovation enables the representation of non-linear bounds with linear constraints.

Recently, Grosen et al's work [2] has advanced AARA, adding support for regular recursive types. This newest iteration of AARA relies on novel resource polynomials to represent potential carried by recursive types. It is capable of deriving bounds on more complex algebraic data structures, such as rose trees, as well as recursive types involving functions.

1.2 Resource Aware ML

Resource Aware ML (RaML) is an implementation of the AARA type system for programs in the ML family (SML, OCaml, etc). It generates resource bounds by leveraging the AARA type system, which generates linear constraints during type inference. For each function in user program, a linear programming (LP) solver attempts to minimize its input resources subject to the constraints — the LP may be infeasible for certain user programs. Otherwise, the LP solution brings us to a final type annotated with a concrete resource bound.

Existing versions of RaML [6] [7] are quite capable. For example, they can derive exact worst-case bounds of resource usage for quicksort, highlighting their ability to tightly analyze tricky recursion patterns. Furthermore, the type-based approach makes it compositional, allowing for easy interprocedural analysis.

RaML does have some shortcomings. For one, it has yet to incorporate Grosen et al's type system (henceforth System GKH), so it does not support regular recursive types. System GKH also takes advantage of another feature lacking from RaML — Kahn et al [9]'s remainder contexts, which allow for the more precise tracking of non-monotone resources, i.e. resources like memory that can become available *during* evaluation. Finally, RaML's implementation has become somewhat overcomplicated and difficult to maintain.

1.3 RaML 2 Outline

To address RaML's shortcomings, we decided to start from scratch instead of modifying existing RaML. The remainder of this thesis is dedicated to presenting this new and improved version of RaML, Resource Aware ML 2, its specific AARA theory, and its implementation in OCaml. It is organized as follows:

- Chapter 2 presents the Intermediate Representation (IR) that the resource analysis operates on, as well as the non-resourceful static semantics and cost aware evaluation semantics. The IR is similar to that of System GKH, with a modification to the elimination form for recursive types.
- Chapter 3 presents the efficient implementation of the IR using an infrastructure known as Abstract Binding Trees.
- Chapter 4 overviews the SML frontend implementation that can translate arbitrary SML code into the RaML IR with the help of the MLton compiler.
- Chapter 5 examines the linear programming (LP) solver backend and some associated utilities which we will invoke to solve linear constraints produced during AARA type inference.
- Chapters 6 and 8 dive into the specific AARA type system of RaML 2, respectively covering resource polynomials and the resource analysis algorithm. Instead of using System GKH verbatim, we develop a type system that can analyze the univariate subset of the possible resource bounds. We also explain how the aforementioned change to the elimination form for recursive types makes the type system more precise.
- Chapters 7 and 9 respectively examine the implementation behind the aforementioned theory of resource polynomials and resource analysis. This includes a significant discussion of RaML's limited support for higher-order functions.
- Chapter 10 showcases RaML 2 analyzing various programs. While this highlights RaML 2's ability to tightly analyze tricky, interprocedural code, as well as tackle rose trees, it also highlights limitations surrounding multivariate analysis.
- Chapter 11 summarizes the results and proposes future improvements for RaML 2.

To illustrate how the pieces of the implementation fit together, here is a diagram of the high-level structure of RaML 2:

Figure 1.1: RaML 2 Structure

Chapter 2

RaML Intermediate Representation

In this chapter, we describe the RaML language, henceforth referred to as the intermediate representation (IR) of RaML 2. This is the language upon which the actual resource analysis will be conducted.

2.1 Language

	Abstract Form	**Concrete Form**	
Type $\tau ::=$	int		
	$\mid \mathsf{float}$		
	$\mid [\ell : \tau_\ell]_{\ell \in L}$		(labelled sum type)
	$\mid \langle \ell : \tau_\ell \rangle_{\ell \in L}$		(labelled product type)
	$\mid \mu t.\ \tau$		(recursive type)
	$\mid \tau_1 \to \tau_2$		(function type)
Expression $e ::=$	$\mathsf{int}\,[n]$	n	
	$\mid \mathsf{float}\,[f]$	f	
	$\mid x$	x	
	$\mid \mathsf{let}\,(e_1, x.\ e_2)$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	(sequential)
	$\mid \mathsf{tick}\,[f]$	$\mathbf{tick}\ f$	(consume/return resources)
	$\mid \mathsf{inj}\,[k]\,\{[\ell : \tau_\ell]_{\ell \in L}\}\,(x)$	$k \cdot x$	(sum introduction)
	$\mid \mathsf{casesum}\,\{\tau\}\,(x, \{\ell : x_\ell.\ e_\ell\}_{\ell \in L})$	$\mathbf{match}\ x\ \mathbf{with}\ [\ell \cdot x_\ell \mapsto e_\ell]_{\ell \in L}$	(sum elimination)
	$\mid \mathsf{tuple}\,(\{x_\ell : x_\ell\}_{\ell \in L})$	$\langle x_\ell : x_\ell \rangle_{\ell \in L}$	(product introduction)
	$\mid \mathsf{caseprod}\,(x, \{\ell : x_\ell\}_{\ell \in L}\,.\ e)$	$\mathbf{match}\ x\ \mathbf{with}\ \langle \ell : x_\ell \rangle_{\ell \in L} \mapsto e$	(product elimination)
	$\mid \mathsf{fold}\,\{t.\tau\}\,(x)$	$\mathbf{fold}\ x$	(recursive type introduction)
	$\mid \mathsf{casefold}\,(x, y.e)$	$\mathbf{match}\ x\ \mathbf{with}\ \mathbf{fold}\ y \mapsto e$	(recursive type elimination)
	$\mid \mathsf{letfun}\,(\{(f : \tau_1 \to \tau_2).x.e, ...\}, f.e')$	$\mathbf{let}\ \mathbf{fun}\ f\ x = e\ \mathbf{and}\ ...\ \mathbf{in}\ e'$	(function(s) introduction)
	$\mid \mathsf{app}\,(f, x)$	$f\ x$	(function elimination)

Figure 2.1: RaML IR Types and Expressions

Figure 2.1 presents the grammar of the RaML IR. For expressions, we present both an abstract and a concrete syntax; the abstract syntax will be used in formalizations like typing rules, while actual example code will follow the concrete syntax. To keep things simpler, we only present one syntax for the types.

Whereas previous iterations of RaML worked with an IR supporting a limited class of recursive types, our IR is a more general language with sums, products, and regular recursive types. In fact, it is essentially just eager FPC with some modifications to facilitate resource analysis. The main modification is that the expressions are in *let-normal form*, meaning we enforce variables instead of general expressions at many sites in the AST; this allows more precise accounting of potential for resource analysis.

Unsurprisingly, our IR is very similar to that of System GKH [2]. There are, however, still some notable differences:

- As this IR is intended to be the basis of an actual implementation, we use labelled n-ary sum and product types instead of pair sums and pair products. This does not impact the high-level theory, but does complicate the typing rules a bit.
- We use casefold instead of unfold as the elimination form for recursive types. The importance of this is explained in chapter 9.
- We have the letfun expression instead of having simple function expressions. This is to support mutually recursive functions.
- Certain expressions are annotated with types. This allows the standard (non-resource-analysis) type inference to be done in a single pass and avoids unification as in Hindley-Milner.
  - inj and fold expressions are annotated with their sum/recursive type.
  - casefold expressions are annotated with their types, since there are no case branches to infer a type from when casing on an empty sum (type $[]$).
  - Each function in a letfun expression is annotated with the argument and result types.

In addition to the presented forms, we have actually implemented throwing and handling exceptions in our IR too. However, since the resource analysis surrounding exceptions is not implemented yet, we do not depict them above.

2.2 Static Semantics

We now present the (non-resourceful) static semantics of the RaML IR. The typing judgement is the standard

$$\Gamma \vdash e : \tau$$

which reads "in the typing context Γ, expression e has type τ". While these rules are subsumed later on by the resourceful typing rules in figure 8.1, we present them for completeness.

$$\frac{}{\Gamma \vdash \mathsf{tick}\,[q] : \langle\rangle}\ \text{T:TICK} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{T:VAR} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma_2, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\,(e_1, x.\, e_2) : \tau}\ \text{T:LET}$$

$$\frac{k \in L}{\Gamma, x : \tau_k \vdash \mathsf{inj}\,[k]\left\{[\ell : \tau_\ell]_{\ell \in L}\right\}(x) : [\ell : \tau_\ell]_{\ell \in L}}\ \text{T:INJ}$$

$$\frac{(\forall \ell \in L) \quad \Gamma, x : [\ell : \tau_\ell]_{\ell \in L}, x_\ell : \tau_\ell \vdash e_\ell : \tau}{\Gamma, x : [\ell : \tau_\ell]_{\ell \in L} \vdash \mathsf{casesum}\,\{\tau\}\left(x, \{\ell : x_\ell.\, e_\ell\}_{\ell \in L}\right) : \tau}\ \text{T:CASESUM}$$

$$\frac{}{\Gamma, x_1 : \tau_1, ..., x_n : \tau_n \vdash \mathsf{tuple}\left(\{\ell_i : x_i\}_{i \in [n]}\right) : \langle \ell_i : \tau_i \rangle_{i \in [n]}}\ \text{T:PROD}$$

$$\frac{\Gamma, x : \langle \ell_i : \tau_i \rangle_{i \in [n]}, x_1 : \tau_1, ..., x_n : \tau_n \vdash e : \tau}{\Gamma, x : \langle \ell_i : \tau_i \rangle_{i \in [n]} \vdash \mathsf{caseprod}\left(x, \{\ell_i : x_i\}_{i \in [n]}.\, e\right) : \tau}\ \text{T:CASEPROD}$$

$$\frac{}{\Gamma, x : [\mu t.\, \tau/t]\tau, \vdash \mathsf{fold}\,\{t.\tau\}\,(x) : \mu t.\, \tau}\ \text{T:FOLD} \qquad \frac{\Gamma, x : \mu t.\, \tau', y : [\mu t.\, \tau'/t]\tau' \vdash e : \tau}{\Gamma, x : \mu t.\, \tau' \vdash \mathsf{casefold}\,(x, y.e) : \tau}\ \text{T:CASEFOLD}$$

$$\frac{\begin{array}{c}\Gamma, f : \tau_1 \to \tau_2, g : \tau_3 \to \tau_4, x : \tau_1 \vdash e_f : \tau_2 \\ \Gamma, f : \tau_1 \to \tau_2, g : \tau_3 \to \tau_4, y : \tau_3 \vdash e_g : \tau_4\end{array} \quad \Gamma, f : \tau_1 \to \tau_2, g : \tau_3 \to \tau_4 \vdash e' : \tau}{\Gamma; p \vdash_c \mathsf{letfun}\,(\{(f : \tau_1 \to \tau_2).x.e_f, (g : \tau_3 \to \tau_4).y.e_g\}, f.g.e') : \tau}\ \text{T:LETFUN}$$

$$\frac{}{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash \mathsf{app}\,(f, x) : \tau_2}\ \text{T:APP}$$

Figure 2.2: RaML IR (Non-Resourceful) Typing Rules

Although these rules are not too interesting, we would like to draw attention to the rules fold and casefold, which describe the behavior of recursive types. The notation $[\tau_1/t]\tau_2$ refers to the capture-avoiding substitution of τ_1 for t in τ_2. To produce an expression of a recursive type $\mu t.\, \tau$, we would first produce an expression with type $[\mu t.\, \tau/t]\tau$, i.e. the "unfolded" version of said recursive type, bind it to some variable x, and wrap it with fold. To use a variable x of a recursive type, we can access its "unfolded" version y and use y in subexpression e via the $\mathsf{casefold}\,(x, y.e)$ construct.

Also, note that letfun can contain any number of mutually recursive functions, and the rule above only has two for brevity.

2.3 Cost Semantics

The cost semantics of the language are standard big-step operational semantics because our expressions are in *let-normal form*. This actually follows how Hoffmann et al's earlier work [7] defines AARA's dynamics instead of System GKH, which opts for small-step operational semantics instead. We start off by defining values:

$$
\begin{array}{rll}
\text{Values } v ::= & \mathsf{int}\,[n] & \\
\mid & \mathsf{float}\,[f] & \\
\mid & k \cdot v & \text{(sum value)} \\
\mid & \langle x_\ell : v_\ell \rangle_{\ell \in L} & \text{(product value)} \\
\mid & \mathbf{fold}\ v & \text{(recursive value)} \\
\mid & \mathbf{fun}\ f \text{ where } f : v \to v & \text{(function value)}
\end{array}
$$

Figure 2.3: RaML IR Values

Note that these values coincide with the introduction form expressions, with the relaxation that variables can be substituted with other values. The only value that does not coincide with a expression is the function, but that is just a result of our letfun construct. Values, like expressions, can be typed with a typing judgement of the form $v : \tau$, whose rules we omit as they are almost a subset of the expression typing rules. This also brings us to the notation $V : \Gamma$, which states that all variables x assigned a type in typing context Γ must be mapped to a value of the same type in evaluation context V, i.e. $V(x) : \Gamma(x)$. Values are not only necessary for the definition of the cost semantics. but will also be the carriers of potential in the AARA theory, which will be explained further in chapter 6.

$$V \vdash e \Downarrow v \mid (q, q')$$

Values are central to the evaluation judgement, which reads "in evaluation context V, expression e evaluates to value v with high-water mark cost q and net cost $(q - q')$". For brevity, we present some simple rules, as well as those surrounding recursive types:

$$\frac{q \geq 0}{V \vdash \mathsf{tick}\,[q] \Downarrow \langle\rangle \mid (q, 0)}\ \text{E:TICK}^+ \qquad \frac{q < 0}{V \vdash \mathsf{tick}\,[q] \Downarrow \langle\rangle \mid (0, -q)}\ \text{E:TICK}^-$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid (q, q') \quad V, x \mapsto v_1 \vdash e_2 \Downarrow v_2 \mid (p, p')}{V \vdash \mathsf{let}\,(e_1, x.\ e_2) \Downarrow v_2 \mid (q, q') \cdot (p, p')}\ \text{E:LET} \quad \text{where } (q, q') \cdot (p, p') = \begin{cases} (q + p - q', p') & \text{if } q' \leq p \\ (q, p' + q' - p) & \text{otherwise} \end{cases}$$

$$\frac{}{V, x \mapsto v \vdash \mathsf{fold}\,\{t.\tau\}\,(x) \Downarrow \mathbf{fold}\ v \mid (0, 0)}\ \text{E:FOLD}$$

$$\frac{V, x \mapsto \mathbf{fold}\ v', y \mapsto v' \vdash e \Downarrow v \mid (q, q')}{V, x \mapsto \mathbf{fold}\ v' \vdash \mathsf{casefold}\,(x, y.e) \Downarrow v \mid (q, q')}\ \text{E:CASEFOLD}$$

Figure 2.4: RaML IR Select Evaluation Rules

tick expressions are the only expressions that directly consume/return resources. let expressions evaluate e_1 to some value v_1 and use that to evaluate e_2 to some value v, and the costs are composed with the defined operator $\cdot$. fold expressions cost nothing on their own, and simply add a **fold** wrapper around the value that the variable within evaluates to. Finally, to evaluate a casefold $(x, y.e)$ expression, we simply evaluate the nested expression e with the extra mapping of y to the "unfolded" version of what x maps to.

Chapter 3

RaML IR Implementation

In this chapter, we describe the implementation of the RaML IR and its associated typing and evaluation.

3.1 Locally Nameless Form

The implementation of the IR is based heavily on Abstract Binding Tree (ABT) infrastructure outlined in Professor Bob Harper's *Practical Foundations of Programming Languages* [4], which also appears in the curriculum of CMU course 15-312, Foundation of Programming Languages. The main benefit of the ABT infrastructure is its ability to represent α-equivalent classes of ASTs. Working up to α-equivalence is very convenient, as it means we never have to deal with variable capture. It also eases operations like comparing terms for equality and substituting terms for variables in other terms.

The ABT infrastructure achieves this by using a more sophisticated representation known as *locally nameless form* (LNF) or *de Bruijn indexed* form. In this representation, each α-equivalence class of terms has exactly one data value — checking for α-equivalence in this representation reduces to structural equality. To better understand, let's take a step back and consider the λ-calculus, whose terms M are either variables x, λ abstractions (functions) $\lambda x.M$, or applications $M_1\ M_2$. Just like in fancier languages like RaML expressions, variables in the λ-calculus serve 2 roles:

1. They can appear free — that is, they can be a name for a yet-to-be-determined term.
2. They can appear bound, in which case the variable occurrences simply refer back to the location of the λ abstraction that bound them.

In a locally nameless representation, we distinguish these two roles in our data structure in order to make α-equivalence implementable as structural equality on terms. The trick in this representation is to exploit the invariant that in any λ abstraction $\lambda x.M$, the only occurrences of x are within M, up to α-equivalence. As a result of this fact, we have a unique path "upward" from each bound variable to the abstraction site that bound it. We can represent this path as a number that tells us how many other abstration sites we have to "hop" over before we reach that bound us. Thus, we can define the following (OCaml) datatype to locally namelessly represent λ-terms:

```
type term =
| BV of int
| FV of variable
| LAM of term
| AP of term * term
```

Listing 3.1: λ-calculus LNF Datatype

Here, BV stands for "bound variable" and stores the (non-negative) number of abstraction sites between the variable and its abstraction, and FV stands for "free variable" and stores an actual variable. For example, let us consider the following λ-term:

$$\lambda x.w\ (\lambda y.(\lambda z.x)(y)(z(\lambda w.y)))$$

Its locally nameless representation would look like

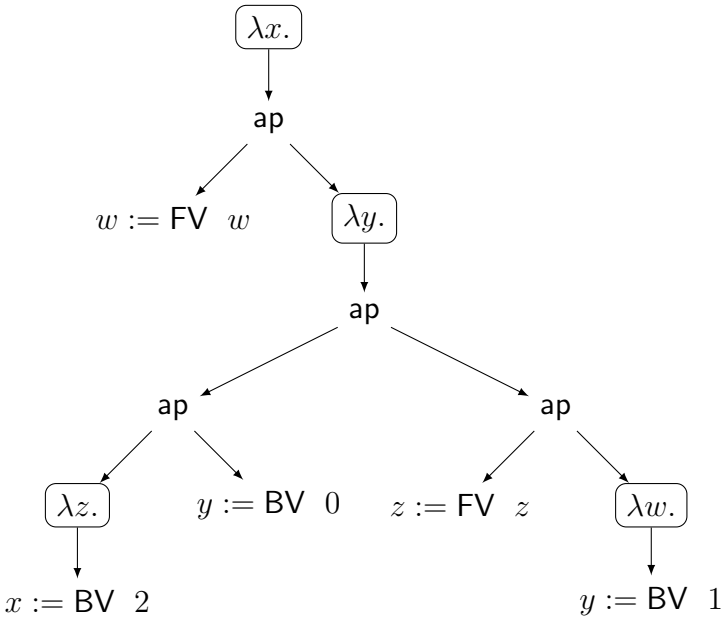

Figure 3.1: λ-calculus LNF Example

In figure 3.1, we distinguish variables as either free or bound, and mark each bound variable with the number of abstractions crossed on the path from itself to its abstraction site. For example, walking "upward" from where x is used to where it is bound crosses over the abstraction sites for y and z, so it is numbered 2. An important fact to notice about these paths is that even for the same binder, each occurrence of its bound variable can have a different bound variable number.

3.2 Abstract Binding Tree Infrastructure

`type term` (defined in listing 3.1) unfortunately only works for the λ-calculus. While we could define a separate LNF datatype for every single sort (type, expression, etc), the much more modular solution that we pursued is to define an Abstract Binding Tree (ABT) functor that can generate the necessary datatypes for each sort. The functor has signature

```
module Make (Var : Var.S) (Oper : Oper.S) : S
```

where `S` is the following interface:

```
module type S = sig
  type t
  include Comparable.S with type t := t

  module Var : Var.S
  module Oper : Oper.S

  type view = Avar of Var.t | Aoper of (Var.t list * t) Oper.t

  val into : view -> t
  val out : t -> view
  val subst : t -> Var.t -> t -> t
end
```

Listing 3.2: Abstract Binding Tree Interface

Instead of granting user code direct access to the locally nameless representation, it is opaque — `type t` is not directly defined in the interface. Instead, users must use the `into` and `out` functions to convert between the transparent datatype `view` and the opaque locally nameless form datatype `t`. The interface also allows for the comparison of locally nameless terms, where equality is simply α-equivalence, as well as the substitution of terms for variables into other terms.

The `view` datatype only has two variants: variable and operator. While this may seem restrictive at first, it becomes clear once we examine what operator is. Recall that in the λ-calculus, terms were either variables, abstractions, or applications. In a fancier language, such as RaML expressions, there are many more syntactic forms other than variables, which we class as operators. Each operator will have varying arrangements of subterms, each with a different valence, or number of abstractions. In the λ-calculus, λ-abstraction $\lambda x.M$ has one subterm, M, with valence 1 corresponding to binder for x, while applications $M_1\ M_2$ have 2 subterms, M_1 and M_2, each with valence 0. In the RaML expressions, the syntactic form let $(e_1, x.\ e_2)$ has 2 subterms, e_1 and e_2, where e_1 has valence 0 and e_2 has valence 1 corresponding to the binder for x.

`Make` takes in two modules as arguments, of which we will focus on `Oper`. This operator module is supplied by the user of the ABT infrastructure, and contains all the aforementioned information about the subterms and their valences for each syntactic form. It has the following signature:

```
module type S = sig
  type 'a t
  include Comparable.S1 with type 'a t := 'a t

  val mapValence : 'a t -> f:(int -> 'a -> 'b) -> 'b t
end
```

Listing 3.3: Operator Interface

In the supplied module, `type 'a t` should be a datatype of all the possible syntactic forms, where occurences of `'a` mark the locations where subterms appear. However, for the ABT functor to correctly access the subterms, we also require the user to supply a `mapValence` function which should call the provided function `f` on each subterm. It also serves the additional purpose of handling valence, which also needs to be passed to `f`. We now present a concrete example in the form of RaML types:

```
type t_base = Bbool | Bint | Bfloat [@@deriving compare, sexp]

module TypOp = struct
  type 'a t =
  | Obase of t_base
  | Oarr of 'a * 'a
  | Osum of 'a Label.Map.t
  | Oprod of 'a Label.Map.t
  | Orec of 'a
  [@@deriving compare, sexp]

  let mapValence abt ~f =
    match abt with
    | Obase bt -> Obase bt
    | Oarr (t1, t2) -> Oarr (f 0 t1, f 0 t2)
    | Osum lmap -> Osum (Label.Map.map lmap ~f:(f 0))
    | Oprod lmap -> Oprod (Label.Map.map lmap ~f:(f 0))
    | Orec t -> Orec (f 1 t)
end

module Abt = Abt.Make (Var.Typ_var) (TypOp)
type t = Abt.t
```

Listing 3.4: RaML IR Type Operator

Observe the type operator `TypOp` supplied to the `Abt.Make` functor. `type 'a t` has a variant for each of the RaML types, and has a `'a` whereever a sub-type (not to be confused with subtyping) appears. For example, operator `Oarr` for function types has two sub-types, which are the τ_1 and τ_2 of function type $\tau_1 \to \tau_2$. More complex subterm patterns are also possible, as seen in operator `Osum`, which takes a map of labels to `'a`'s. As for `mapValence` observe how it calls `f` on the sub-types in each variant. The valence arguments is 0 for all calls except the one for recursive types, since in the recursive type $\mu t.\ \tau$, the sub-type τ has valence of 1, having a binder for the type variable t. The resulting module `Abt` contains the LNF datatype for RaML IR types, which we extract via `type t = Abt.t`.

Now let's go back to the ABT interface from listing 3.2 to understand how to actually work with values of type `Abt.t`. Observe how `Abt.view` never exposes the underlying bound variable infrastructure to the users. Instead, looking at `Aoper`, each subterm `t` is packed with a `Var.t list`. When the `Abt.out` function is applied to a term that includes variable binders, it uses the `Var` module to generate a fresh variable for each bound variable, and substitutes that newly free variable for the bound variable in the subterm. `Abt.into` achieves the reverse. Taking recursive types as an example,

- Suppose `rectyp : Abt.t` is the LNF representation of IR recursive type $\mu t.\ \tau$. Then `Abt.out rectyp` will return `Aoper (Orec ([t], typ))`, where `t : Var.t` is some fresh variable x, and `typ : Abt.t` is the LNF representation of $[x/t]\tau$.
- Suppose `t : Var.t` represents some variable x and `typ : Abt.t` is the LNF representation of some type τ which may use variable x. Then `Abt.into Aoper (Orec ([t], typ))` will return `rectype : Abt.t` which is the LNF representation of IR recursive type $\mu x.\ \tau$.

3.3 Types and Expressions

While the modules produced by the `Abt.Make` functor are sufficient to implement types and expressions for the RaML IR, they are a bit cumbersome to work with. To pattern match on an IR type for example, we need to first call `Abt.out`, pattern match on whether it is a variable `Avar` or an operator `Aoper`, and then pattern match against the underlying operator. To streamline this process, we have added an additional layer of interface/implementation separation to conceal the machinery. RaML types have the following interface:

```
type var = Var.Typ_var.t
type t
include Comparable.S with type t := t

type t_base = Bbool | Bint | Bfloat
type view =
| Tvar of var
| Tbase of t_base
| Tarr of t * t
| Tsum of t Label.Map.t
| Tprod of t Label.Map.t
| Trec of (var * t)

val into : view -> t
val out : t -> view
val subst : t -> var -> t -> t
```

Listing 3.5: RaML IR Type Interface

The datatype `Typ.view` now does not need to expose the fact that the additional operator `Aoper` layer from the `Abt` module. The helper functions `Typ.into` and `Typ.out` respectively invoke `Abt.into` and `Abt.out`, along with some pattern matching, converting between `Typ.view`'s and their underlying LNF representations.

```
type var = Var.Exp_var.t
type t
include Comparable.S with type t := t

type func = { typ : Typ.t; func : var; arg : var; body : t }
type view =
| Evar of var
| Eletvar of { e1 : t; x : var; e2 : t }
| Eletfun of { funs : func list; e : t }
...
| Efold of { typ : Typ.t; arg : var }
| Ecasefold of { arg : var; unfolded : var; body : t }

val into : view -> t
val out : t -> view
```

Listing 3.6: RaML IR Expression Interface

RaML IR expressions have a similar interface. For brevity, we have omitted certain `Exp.view` variants and additional helpers. However, the omission of `subst` for substitution is intentional. An unfortunate side effect of *let-normal form* syntax of expressions is that substitution of expressions for variables in other expressions is not always well-formed.

3.4 Semantics

In addition to the IR itself, we also implemented its semantics. For the static semantics, we implemented the standard non-resourceful typing rules presented in figure 2.2. Owing to the type annotations we added for injections, cases, and functions, type inference of any expression is possible with a single pass, eliding the need for type unification.

```
module StaticsError : sig
  type t = ...
  val to_string : t -> string
end

exception TypeError of StaticsError.t

val validateType : Typ.t -> unit
val inferType : Exp.t -> Typ.t
```

Listing 3.7: RaML IR Statics Interface

The statics interface is quite simple. `validateType` checks that a given type does not have any unbound (free) type variables. `inferType` implements the typing rules, and outputs the overall type of a given expression. If an expression is not well-typed, `inferType` will throw a **`StaticsError`** that can be handled by the caller. Note that the typing context Γ is not required as an argument, as we only type closed expressions, i.e. those typed with an empty context.

```
module Cost : sig
  type t
  ...
  val get : t -> float * float
end

type value =
| Const of Exp.t_base
| Lam of (value -> Cost.t * value)
| Fold of value
| Inj of Label.t * value
| Prod of value Label.Map.t

module DynamicsError : sig
  type t = ...
  val to_string : t -> string
end

exception Malformed of DynamicsError.t

val eval : 'a Exp.t -> Cost.t * value
```

Listing 3.8: RaML IR Cost Semantics Interface

The cost semantics also have a straightforward implementation following their rules, partially presented in figure 2.4. `eval` implements the evaluation rules, returning the value that an expression evaluates to, alongside its cost. Observe that instead of using a subset of expressions to represent RaML values, we define a datatype **`value`**. Although the type safety theorems should ensure that any expression that typechecks should evaluate without issues, `eval` will throw a **`DynamicsError`** if the provided expression is in fact malformed.

Chapter 4

SML Frontend

In this chapter, we examine the frontend of the RaML 2 implementation. Instead of creating a frontend directly for the RaML IR, we have implemented a frontend that can take SML code and transpile it into the RaML IR.

4.1 OCaml Frontend?

Previous iterations of RaML had an OCaml frontend instead of an SML one. There are certainly several reasons to favor an OCaml frontend. For one, OCaml is a far more popular language than SML, so RaML 2 would be more accessible to the public with an OCaml frontend. More importantly, since RaML 2 is implemented in OCaml, it has access to the `ocaml-compiler-libs`, which would allow it to directly access the OCaml AST, as well the ASTs of the IRs within the OCaml compiler.

Despite these advantages, the shortcomings of the OCaml compiler ultimately convinced us to implement an SML frontend instead. The OCaml compiler is not very incremental, translating OCaml source code directly to a low level language, Lambda. For our frontend to plug into the OCaml compiler, it would either have to translate the OCaml source code or the Lambda language into our IR. Translating from OCaml source would require complex transformations like flattening nested pattern matches, as well as type inference, since our IR requires type annotations. On the other hand, translating from Lambda is even more impossible, as it has already destroyed higher level constructs like sum and recursive types.

SML's MLton compiler, on the other hand, is far more incremental, given us a perfect place to plug into and translate to the RaML IR. The rest of this chapter presents our SML frontend that uses MLton.

4.2 Overview

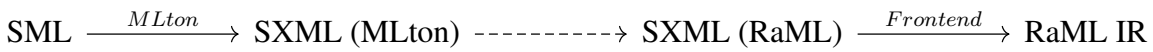

Figure 4.1: Frontend Structure

The frontend broadly consists of two pieces. We first invoke the MLton compiler; halfway through the MLton pipeline is an internal MLton IR named SXML, which we output into a file. RaML then picks up here, parsing the SXML code and transpiling it into the RaML IR. The unfortunate redundant printing and parsing of SXML in the middle of our frontend pipeline arises because the MLton compiler is written in SML while RaML is written in OCaml. We ultimately deemed the advantages conferred by MLton to be worth the extra complexity of the frontend.

4.3 MLton

MLton is a whole-program optimizing compiler for SML. Whereas SML/NJ or the OCaml compiler jumps straight from a high-level IR close to the source language straight to a low-level IR close to native code, MLton compilation is incremental, passing through far more Intermediate Representations (IRs). We specifically chose to use MLton for the first half of the frontend because one of its IRs, SXML (explored further in section 4.4), is quite close in structure to the RaML IR, making translation a much more tractable problem.

Taking a step back though, let's go over what exactly MLton [1] does for us. SML may be a frugal language, but compared to the RaML IR, it still has many more bells and whistles — features need to be reduced to simpler constructs.

$$\text{SML} \xrightarrow{Elaborate} \text{CoreML} \xrightarrow{Defunctorize} \text{XML} \xrightarrow{Monomorphise} \text{SXML}$$

Figure 4.2: MLton Frontend Structure

1. *Elaborate*: After type inferencing/checking the original SML code, this pass then defunctorizes the program, lifting everything within modules into the top level of the program. This defunctorized IR is named CoreML. This pass is necessary because the RaML IR does not support modules.
2. *Defunctorize*: Contrary to its name, this pass no longer defunctorizes, which is now done by the previous *Elaborate* pass. Among other things, the translation from CoreML to XML converts all case expressions with nested patterns into case expressions with flat patterns in a subpass *MatchCompile*. Flattening patterns is necessary because the RaML IR can only pattern match at top level pattern constructor. For example, consider the following list identity function in SML:

   ```
   fun id1 l =
     case l of
       [] => []
     | x::xs => x::(id1 xs)
   ```

   Even this very simple code contains a nested pattern match; the second case not only matches for the list cons (::) operator, but also that its associated data is a pair. The flattened version of the pattern match would be:

   ```
   fun id1 l =
     case l of
       [] => []
     | :: m =>
       case m of
         (x, xs) => x::(id1 xs)
   ```

   In fact, the presence of *MatchCompile* in MLton was one of the primary factors for its adoption for this project, as handwriting code to correctly flatten patterns was a significant undertaking during the implementation of the original RaML.
3. *Monomorphise*: This pass is relatively simple, and accurate to its name, monomorphizes XML into SXML by duplicating all polymorphic values and types for each type at which they are used. It is necessary to include, as RaML cannot work directly with polymorphic expressions.

In addition to these translation passes, we also take advantage of some MLton optimization passes on the CoreML and XML IRs. These passes conduct helpful reductions to further simplify the code, such as but not limited to

- `let x = e1 in e2` to `[e1/x]e2` if `x` occurs 0 or 1 times in `e2`
- `(fn x => e1) e2` to `let x = e1 in e2`

The translation passes, especially *MatchCompile*, often introduce functions into the code that are only called once, so these optimization passes help to inline said functions, greatly benefiting the RaML analysis down the line.

4.4 XML (SXML) Language

The SXML language forms the juncture between the MLton and handwritten parts of the RaML frontend. While SXML is the monomorphized version of XML, both actually use the exact same AST, so we will henceforth just call it XML.

XML is still rather high-level and similar to SML in many ways. It uses SML-style datatypes, i.e. equirecursive labelled sum types. It still contains higher-order functions. While some importance differences are already stated in the previous section, we reiterate them here, along with some further differences.

- XML only has flat pattern matching, instead of nested pattern matching. This is explained in the previous section.
- XML only has unlabelled products, i.e. tuples, instead of records. XML tuples are eliminated via projections, i.e. $p.2$ to get the 3rd entry of tuple p.
- XML is in *let-normal form* like RaML. That is, the arguments for all syntactic forms are already variables.
- MLton has annotated each XML value declaration with its type, as in `let val x : typ = ...`. This means we do not need to do any type inference when translating to RaML expressions with type annotations.

The RaML handwritten frontend begins by reading in XML from a file generated by MLton. The raw input is parsed into an AST with an OCaml parser combinator library named *Angstrom*.

4.5 Dead Code Elimination

Before the actual translation to the RaML IR, we run a dead code elimination pass on XML. This is not as superfluous as it sounds; the MLton compiler includes everything from included libraries into the XML output code. Even a very simple piece of user code will generate an XML file with several tens of thousands of lines, due to the presence of values and datatypes included from the SML Basis Library.

Thus, to minimize the code we need to translate to RaML, we eliminate any unneeded or dead code with a mark-and-sweep style algorithm:

1. Mark all user declarations as *needed*.
2. Propagate *needed*-ness by finding all free variables and types used in currently *needed* declarations, and marking the declarations of these variables and types as *needed*.
3. Repeat the previous step until saturation.
4. Delete, i.e. sweep away, all un-*needed* declarations.

4.6 Translation to RaML IR

XML is somewhat similar to the RaML IR, and most of its syntactic forms can be directly translated into the RaML IR. Furthermore, it is already in let-normal form, so we do not need to deal with that. There are however several features that we need to translate with care, which we will focus on here.

4.6.1 Types

Most XML types have direct analogues in the RaML IR. The notable exception is datatypes. Let $\mathsf{translate}(\tau, \Sigma)$ denote the translation of an XML type τ into RaML IR, where the argument Σ is a function that translates datatype names into RaML types. Here are some select cases of the translation:

$$\begin{aligned}
\mathsf{translate}(\mathsf{word64}, \Sigma) &= \mathsf{int} \\
\mathsf{translate}(\mathsf{arrow}(\tau_1, \tau_2), \Sigma) &= \mathsf{translate}(\tau_1, \Sigma) \to \mathsf{translate}(\tau_1, \Sigma) \\
\mathsf{translate}(\mathsf{tuple}(\tau_1, ..., \tau_n), \Sigma) &= \langle \ell_i : \mathsf{translate}(\tau_i, \Sigma) \rangle_{i \in [n]} \\
\mathsf{translate}(d, \Sigma) &= \Sigma(d)
\end{aligned}$$

This translation is straightforward for the most part, but we are clearly offloading significant effort onto Σ for the translation of datatypes, which we explain in the next section.

4.6.2 Datatypes

XML datatypes, much like SML ones, are equirecursive rather than isorecursive. This means that unlike the RaML IR's recursive types, there is neither an explicit type constructor for recursive types, nor fold and unfold expressions. Furthermore, while XML equirecursive datatypes can easily support mutual recursion, the RaML IR does not. Finally, since datatypes and types in XML can include each other, their translation processes are mutually recursive too, which is already hinted at above by the inclusion of Σ into the translation for XML types.

This brings us to Σ_0, which will denote as the canonical datatype translator. This canonical translator will be the Σ used to translate the types in the body of the program as follows:

$$\mathsf{translate}'(\tau) = \mathsf{translate}(\tau, \Sigma_0)$$

Let Γ be a lookup map that finds the constructors of some datatype d and their corresponding XML types. Then the pseudocode for Σ_0 is

$$\begin{aligned}
&\Sigma_0(d) = \\
&\quad [\ell : \tau_\ell]_{\ell \in L} \leftarrow \Gamma(d); \\
&\quad \mathsf{fresh}\ t; \\
&\quad \Sigma(d') = \mathsf{if}\ d = d'\ \mathsf{then}\ t\ \mathsf{else}\ \Sigma_0(d'); \\
&\quad \mu t.\ [\ell : \mathsf{translate}(\tau_l, \Sigma)]_{\ell \in L}
\end{aligned}$$

Essentially, $\Sigma_0(d)$ always outputs a RaML recursive type wrapped around a sum type, where the summands are the constructor types translated from XML. The catch is that the XML types are translated with a datatype translator that is not merely Σ_0; instead, Σ will output the type variable t when it sees the datatype d, only calling Σ_0 otherwise. This ensures that when translating some datatype d, if we ever see d occur within its constructor types, the translation does not loop forever, and correctly marks the recursion site.

For example, the int list datatype

```
datatype list = nil | :: of (int, list)
```

will be correctly translated by Σ_0 into the RaML IR type of

$$\mu t.\, [\mathsf{nil} : \langle\rangle\,, \mathsf{cons} : \langle \mathsf{int}, t\rangle]$$

Unfortunately, Σ_0 alone fails on mutually recursive datatypes. Consider int rose trees defined in XML datatypes:

```
datatype rose = Rose of int * rose_list
datatype rose_list = nil | :: of (rose, rose_list)
```

Applying the Σ_0 translation separately to both `rose` and `rose_list` will result in

$$\begin{aligned}\text{rose} &:= \mu a.\, \left[\mathsf{Rose} : \left\langle \mathsf{int}, \mu b.\, [\mathsf{nil} : \langle\rangle\,, \mathsf{cons} : \langle a, b\rangle]\right\rangle\right] \\ \text{rose\_list} &:= \mu b.\, \left[\mathsf{nil} : \langle\rangle\,, \mathsf{cons} : \left\langle \mu a.\, [\mathsf{Rose} : \langle \mathsf{int}, b\rangle], b\right\rangle\right]\end{aligned}$$

While these two translations are valid on their own, translating every `rose` and `rose_list` accordingly in the same program will create type errors. To understand how, consider the following IR expression when given a value x of type rose.

$$\mathbf{match}\ x\ \mathbf{with\ fold}\ y \mapsto \left(\mathbf{match}\ y\ \mathbf{with}\ \ [\mathsf{Rose} \cdot p \mapsto \mathbf{match}\ p\ \mathbf{with}\ \ \langle i, rs\rangle \mapsto rs]\right)$$

Given a rose tree value, this expression extracts the list of children trees. Applying the standard typing rules from figure 2.2, we get that this expression has type

$$\text{rose\_list}' := \mu b.\, \left[\mathsf{nil} : \langle\rangle\,, \mathsf{cons} : \left\langle \mu a.\, \left[\mathsf{Rose} : \left\langle \mathsf{int}, \mu b'.\, [\mathsf{nil} : \langle\rangle\,, \mathsf{cons} : \langle a, b'\rangle]\right\rangle\right], b\right\rangle\right]$$

Although this type rose_list$'$ is isomorphic to rose_list from above, our type system works with recursive types at a more syntactic level, so they are considered different types. Thus, if we translated every XML `rose_list` to an IR rose_list, the translated program would not typecheck. Instead, we must translate every XML `rose_list` to an IR rose_list$'$ for everything to be well-typed.

The general solution to translating mutually recursive datatypes begins by first partitioning all the XML datatypes into strongly connected components (SCCs) per their dependency graph. Within the dependency graph, there is a directed edge from datatype a to b if datatype b is used as a type within one of the constructors for datatype a. Thus, each SCC consists of datatypes that all strongly mutually recurse with each other.

Next, we apply the Σ_0 datatype translator to an arbitrary datatype within each SCC. For example, in our rose tree example above, datatypes `rose` and `rose_list` form an SCC, and we arbitrarily applied the Σ_0 translation to `rose` to get the IR type rose. Finally, starting from this initial translation, we can construct the correct translations for the other datatypes in the SCC by destructing the initial translation to find instances of the other datatypes in the SCC. This will require traversing the dependency graph within the SCC. For our rose tree example, this is just how we constructed rose_list$'$ above by destructing a rose.

Finally, after every XML datatype has an IR type translation, we make another pass over all of them to remove unneeded recursive type abstractions. For example, this pass ensures that booleans are translated as the sum type $[\mathsf{true} : \langle\rangle\,, \mathsf{bool} : \langle\rangle]$ without an unnecessary recursive type $\mu t.$ prefix.

4.6.3 Injections and Cases

Injections and case expressions need to be similarly translated to work properly with recursive types. Specifically, if an XML injection expression's translated type is a recursive type, then its corresponding RaML expression must actually be a fold wrapped around an injection. Likewise, case expressions in XML need to be translated to a casefold expression wrapped around a casesum expression. For example, the XML list expression `3 :: xs` will be translated into

$$\mathsf{fold}\ \{t.[\mathsf{nil} : \langle\rangle\,, \mathsf{cons} : \langle \mathsf{int}, t\rangle]\}\ (\mathsf{cons} \cdot \langle 3, xs\rangle)$$

4.6.4 Projections

While MLton's translation to XML very helpfully flattens nested patterns, something less helpful it also does is replacing tuple pattern matches with projection expressions instead. The RaML IR eliminates tuples with pattern matches however, so we essentially need to undo this part of the translation when going from XML to the RaML IR.

To do this, our translation stores a context Δ, where given $x : \langle \ell : \tau_\ell \rangle_{\ell \in L}$, $\Delta(x)$ is a map $\{\ell : x_\ell\}_{\ell \in L}$ of each projection label to a variable representing that tuple entry. The first time a tuple variable x is projected from, we generate a new entry in Δ with fresh variables for each x_ℓ, and insert a caseprod expression into the translated output. All subsequent projections from the same tuple x will then be replaced with the corresponding x_ℓ.

For example,

```
val a = p.x
val b = f a
val c = p.y
...
```

will be translated to

$$\begin{aligned} \mathsf{caseprod}\ p\ \mathsf{of}\ (x, y) \Rightarrow\ & \mathsf{let}\ a = x\ \mathsf{in} \\ & \mathsf{let}\ b = f\ a\ \mathsf{in} \\ & \mathsf{let}\ c = y\ \mathsf{in} \\ & \ldots \end{aligned}$$

4.6.5 Ticks

Ticks are exclusively a RaML feature, and have no direct equivalent in XML or SML. Our solution to this is to define a `Raml` module that needs to be included with any user program. This module contains a dummy `tick` function of type `real -> unit`, so that user SML programs can use it in a manner as close to RaML as possible.

When our translation process is translating XML function applications, most are translated to RaML IR function applications. The exception is when the function is `Raml.tick`, in which case our translation outputs the RaML tick expression.

4.7 Example

```
fun append (a, b) =
  case a of
    [] => b
  | x :: xs => (Raml.tick 4.11; x :: (append (xs, b)))
```

```
datatype list = rec(t.[:: -> (int * t), nil -> ()])

let rec append (x_14242_242 : list * list) : list =
  match x_14242_242 with p_412, p_413 -> (* caseprod *)
    match p_412 with fold temp_414 -> (* casefold *)
      match[list] temp_414 with (* casesum *)
      | :: x_14244_415 ->
        match x_14244_415 with p_417, p_418 -> (* caseprod *)
          let x_14248_419 = tick(4.110000) in
          let x_14249_420 = (p_418, p_413) in
          let x_14250_421 = append x_14249_420 in
          let x_14251_422 = (p_417, x_14250_421) in
          let temp_423 = :: x_14251_422 in fold{list}(temp_423)
      | nil unused_416 -> p_413
```

Listing 4.1: Frontend Translation of Append

We now present an example result of our SML frontend in action. The top half of the listing is a simple list append function in SML which calls tick every recursive call. It is translated to the RaML IR code below, albeit with a slightly different concrete syntax. Observe how the SML **case** expression gets dissected into 3 different matches in the RaML IR: casefold to unwrap the recursive binder for lists, casesum to actually case on whether the list is nil or cons, and finally a caseprod in the cons case to separate the head and tail. Also, note that everything has been converted to let-normal form.

Chapter 5

Linear Programming Backend

In this chapter, we will examine the implementation of the linear programming (LP) backend. The core resource analysis algorithm of RaML relies on generating linear constraints and solving them with an LP solver.

5.1 LP Solver Interface

While we have currently implemented a backend for the COIN-OR Linear Program (CLP) solver, we plan to implement additional solvers in the future, as well as allowing users to register their own LP backends. To streamline the interaction of RaML with these LP backends, we have devised the following interface for LP solvers:

```
module type S = sig
  module VMap : Core.Map.S

  exception E of string

  type var = VMap.Key.t

  val clear_all : unit -> unit
  val fresh_var : unit -> var
  val var_to_string : var -> string
  val add_constr : ?lower:float -> ?upper:float -> float VMap.t -> unit
  val set_objective : float VMap.t -> unit
  val solve : unit -> result
  val get_objective_val : unit -> float
  val get_solution : var -> float
  val get_num_constraints : unit -> int
  val get_num_vars : unit -> int
end
```

Listing 5.1: LP Solver Interface

The interface is imperative due to the formulation of the LP problem, and the design of most LP solvers. There is a hidden state in each solver instance containing the current LP variables, constraints, and objective function. The methods of the interface modify this state, and `clear_all` resets it, clearing all data.

The `VMap` module represents a map where the keys are LP variables. This is used in the two key LP solver operations

- `add_constr` lower upper $\{x_1 : a_1, ..., x_n : a_n\}$ adds the linear constraint

$$\text{lower} \leq a_1 x_1 + \ldots + a_n x_n \leq \text{upper}$$

- `set_objective` $\{x_1 : a_1, ..., x_n : a_n\}$ sets the objective function that is being minimized to

$$f(\vec{x}) = a_1 x_1 + \ldots + a_n x_n$$

The other operations are self explanatory.

5.2 COIN-OR Linear Program Solver Backend

As mentioned, we have thus far implemented an actual backend for CLP. This backend calls on CLP functions through *C* Programming Language stubs, which can be invoked from RaML via OCaml's foreign function interface. The implementation of the solver interface using the CLP stubs is mostly straightforward, with a few interesting kinks.

```
let fresh_var () =
  let count = !var_count in
  if count % col_buffer_size = 0 then Stubs.add_columns !clp_state many_cols
  ↪  col_buffer_size;
  var_count := count + 1;
  count
```

Listing 5.2: CLP Fresh Variable Implementation

For one, `fresh_var` (shown above) and `add_constr` actually buffer new variables/constraints to be added, and only invoke the *C* stubs once their respective buffers are filled. This ensures that even when many variables and constraints are generated during RaML analysis, the foreign function interface is not repeatedly invoked, reducing overhead.

Another matter of note is the `solve` function. CLP is able to efficiently solve an LP system incrementally. This is very useful for RaML, where there are multiple instances where we need to solve a system, add more constraints, then solve again. The very first call to `solve` actually invokes a CLP function `Clp_initialSolve`, while all future calls invoke `Clp_dual`, until the solver function `clear_all` resets this.

5.3 Solver Utility Module

On top of the LP backend, we also implement a helpful utility module. The point of this module is to enable easy manipulation of linear combinations of LP variables. This will prove invaluable to simplifying implementation of resource annotations in chapter 6.3. Here is the interface

```
module type S = sig
  module Solver : Lp.Solver.S

  type t

  val float : float -> t
  val generate : unit -> t

  val ( + ) : t -> t -> t
  val ( <= ) : t -> t -> unit
  val ( >= ) : t -> t -> unit

  val minimize : t -> unit
  val get_value : t -> float

  val to_string : t -> string
end
```

Listing 5.3: LP Solver Utility Interface

The underlying type of this signature is

```
type t = { vmap : float VMap.t; const : float }
```

`vmap` maps each LP variable to its coefficient, while `const` is the additional constant value. Namely, the linear combination $a_1x_1 + \ldots a_nx_n + c$ is represented as $\mathsf{vmap} = \{x_1 : a_1, ..., x_n : a_n\}, \mathsf{const} = c$.

`float` and `generate` are used to generate "seed" linear combinations. The former generates one consisting of only the constant offset, and the latter generates one consisting of only a single LP variable, via `Solver.fresh_var` (see listing 5.2).

The `( + )` operator simply adds two linear combinations by adding coefficients if an LP variable occurs in both input maps. The `( <= )` operator imperatively appends a new linear constraint to the LP `Solver` instance (via `Solver.add_constr`) which enforces that its first linear combination argument is less than the second.

Finally `minimize` invokes the LP solver functions `set_objective` and `solve` to try to minimize the input linear combination, then calls `get_objective_val` to obtain the best value it can attain. It then actually inserts a new constraint to enforce that this specific linear combination must be less than the attained objective value, requiring that future solves on this current LP instance respect this. The importance of this will be explained in chapter 7.

Chapter 6

Resource Polynomials

In this chapter, we present univariate AARA resource polynomials that the RaML type system, discussed in chapter 8, is based on. Be warned that it is not self-contained and relies heavily on prior knowledge of System GKH [2].

6.1 Base Polynomial Indices

The original RaML was devised to primarily analyze programs that worked with lists, as opposed to general data structures. Its univariate version could represent the polynomial potential functions involved in working with lists by annotating types with potential information at strategic locations. For example, $L^{(q_0,q_1,\ldots,q_n)}(A)$ is the annotated type for a list of A's, with q_0 constant potential, q_1 linear potential, q_2 quadratic potential, etc. The coefficients q_i correspond to basis of binomial coefficients $\binom{n}{i}$ where the list is of length n.

Multivariate AARA presented some new challenges. Polynomial potential functions could now mix products of multiple variables, which is necessary, for example, when analyzing code that computes the Cartesian product of two lists. To address this, multivariate AARA introduced the concept of *indices* to form a basis of potential functions. Indices disentangle resource annotations from types by essentially "naming" base polynomial potential functions. Intuitively, we can view an index as a pattern which we match against a value (see figure 2.3), counting the number of occurrences of a certain "shape" in the value. This will be clarified later in the section.

While we only implemented univariate AARA, we decided to incorporate indices into our implementation. For one, our implementation supports regular recursive types and not just lists, for which indices make the implementation more uniform and elegant. Additionally, we do plan to implement multivariate AARA eventually, so implementing an index system now is good preparation.

The indices of our univariate AARA with recursive types are inspired by System GKH's indices for multivariate AARA with recursive types. Let $\mathcal{I}(\tau)$ be the set of indices for a given type τ. Figure 6.1 shows the inference rules for the judgement $i \in \mathcal{I}(\tau)$.

$$\frac{k \in L \quad i \in \mathcal{I}(\tau_k)}{\mathsf{inj}_k \cdot i \in \mathcal{I}([\ell : \tau_\ell]_{\ell \in L})} \qquad \frac{k \in L}{\mathsf{inj}_k^* \in \mathcal{I}([\ell : \tau_\ell]_{\ell \in L})} \qquad \frac{k \in L \quad i \in \mathcal{I}(\tau_k)}{\mathsf{prj}_k \cdot i \in \mathcal{I}(\langle \ell : \tau_\ell \rangle_{\ell \in L})} \qquad \frac{i \in \mathcal{I}([\mu t.\ \tau/t]\tau)}{\mathsf{fold} \cdot i \in \mathcal{I}(\mu t.\ \tau)}$$

Figure 6.1: Index Set Inference Rules

Our indices do have some notable differences with the multivariate System GKH, however:

- In univariate AARA, it makes sense to keep the constant potential separate from the annotated context. (This will become clearer in chapter 8's the resource typing judgement.) This means we no longer have the constant indices for functions and recursive types.
- The indices for product types have become simplified, and look very similar to those of sum types. This is because univariate AARA cannot mix potential between the entries of a product, so the indices for each entry are entirely disjoint from those of the other entries.
- We are not able to fully disentangle constant potential attached to sum types, because each injection can carry a different amounts of potential. As a result, we introduce new indices inj_k^* for the sum types representing the constant potential carried by each injection k of the sum. In fact, each index must "end" in one of these new indices, as they are the only base case index.

For example, the indices of a list of integers, i.e. the type $\mu t.\ [\mathsf{nil} : \langle\rangle\ , \mathsf{cons} : \langle \mathsf{head} : \mathsf{int}, \mathsf{tail} : t\rangle]$, are

$$\begin{aligned}
&\mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{nil}},\\
&\mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}},\\
&\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{tail}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{nil}},\\
&\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{tail}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}},\\
&\ldots
\end{aligned}$$

There is a surjection of these indices onto the coefficients $\vec{q}$ of the old-style annotated type $L^{(q_0,q_1,\ldots,q_n)}(A)$. For example, the constant potential carried on a cons injection of the list, named by the index $\mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}}$, is precisely q_1, the potential carried by each element. Likewise, quadratic potential q_2 lines up with our index $\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{tail}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}}$, which counts all ordered pairs of cons injections, of which there are $\binom{n}{2}$ of. The indices ending in $\mathsf{inj}^*_{\mathsf{nil}}$ are extraneous for lists because each list contains exactly one nil injection anywhere in it. However, it would be incorrect to remove them, as this is just a special case.

A more general case that better illustrates the generalized nature of these indices is rose trees. A rose tree of integers, i.e. the type $\mu a.\ \left[\mathsf{Rose} : \left\langle \mathsf{key} : \mathsf{int}, \mathsf{kids} : \mu b.\ [\mathsf{nil} : \langle\rangle\ , \mathsf{cons} : \langle \mathsf{head} : a, \mathsf{tail} : b\rangle]\right\rangle\right]$, would have the indices

$$\begin{aligned}
&\mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{Rose}},\\
&\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{Rose}} \cdot \mathsf{prj}_{\mathsf{kids}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{nil}},\\
&\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{Rose}} \cdot \mathsf{prj}_{\mathsf{kids}} \cdot \mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{head}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{Rose}},\\
&\ldots
\end{aligned}$$

The first index corresponds to the potential carried by each node of the rose tree. The second index is unfortunately redundant again; it refers to the potential carried by each nil injection appearing in the children list of any node, but since each list always has exactly one nil injection and each node contains a children list, it coincides with the first index. The third index shown encodes far more interesting properties. It represents the potential carried by an ordered pair consisting of a rose tree node and one of its descendant nodes.

6.2 Base Polynomial Evaluation

While indices are how we name base polynomial potential functions, we need a way to actually measure the potential on concrete values. This brings us to the next building block: index evaluation function $\phi_i(v : \tau)$. The index evaluation function measures the potential carried by the value v specifically with regards to index i. In fact, it formalizes the previously mentioned notion of "counting" the number of matches of index i in v.

$$\phi_{\mathsf{inj}_k \cdot i}(\ell \cdot v : [\ell : \tau_\ell]_{\ell \in L}) = \begin{cases} \phi_i(v : \tau_k) & \text{if } l = k \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_{\mathsf{inj}_k^*}(\ell \cdot v : [\ell : \tau_\ell]_{\ell \in L}) = \begin{cases} 1 & \text{if } l = k \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_{\mathsf{prj}_k \cdot i}(\langle \ell : v_\ell \rangle_{\ell \in L} : \langle \ell : \tau_\ell \rangle_{\ell \in L}) = \phi_i(v_k : \tau_k)$$

$$\phi_{\mathsf{fold} \cdot i}(\mathbf{fold}\ v : \mu t.\ \tau) = \phi_i(v : [\mu t.\ \tau/t]\tau) + \sum_{k \in \mathcal{M}\{t.\tau\}(\mathsf{fold} \cdot i)} \phi_k(v : [\mu t.\ \tau/t]\tau)$$

Figure 6.2: Index Evaluation Function

As expected, the index evaluation function for univariate AARA is somewhat similar to System GKH's index evaluation function, with differences arising due to our choice of indices. Some things to note:

- Only values whose types contain a sum type within can be evaluated against an index. All other values/types, such as functions, carry no potential, so they have no indices for us to evaluate against.
- Recalling our indices for product types, note that we lose the nice multiplication of evaluation on multiple indices present in System GKH's multivariate index evaluation for product types.
- The $\mathcal{M}\{t.\tau\}(i)$ recursive occurences function comes directly from System GKH, and it returns the set of indices corresponding to placing index i at every occurrence of t in τ.

The index evaluation function is not itself required for resource analysis, but is useful as a building block for theorems about the analysis algorithm and in checking the resource analysis results against the cost semantics.

6.3 Resource Annotations

Formally, as described by Grosen et al [2], resource annotations $P, Q \in \mathcal{A}(\tau)$ for a specific type τ are elements of the $\mathbb{Q}_{\geq 0}$ semimodule with $\mathcal{I}(\tau)$ as its basis. This means $\mathcal{A}(\tau)$ is like a vector space with $\mathcal{I}(\tau)$ as its basis, and supports operations

- $+ : \mathcal{A}(\tau) \times \mathcal{A}(\tau) \to \mathcal{A}(\tau)$ that satisfies commutative monoid laws
- $\cdot : \mathbb{Q}_{\geq 0} \times \mathcal{A}(\tau) \to \mathcal{A}(\tau)$ that satisfies identity laws and distributes over $+$

More intuitively, an annotation for τ maps every index in $\mathcal{I}(\tau)$ to a positive rational coefficient, and can be added and scaled in expected ways.

First, we extend index evaluation functions to potential functions evaluating annotations. Letting P be an annotation and p_i be the rational coefficient that index i maps to,

$$\Phi_P(v : \tau) = \sum_{i \in \mathcal{I}(\tau)} p_i \phi_i(v : \tau)$$

This is the full potential function from the physicist's method of amortized analysis that we use on data structures to get how much potential they carry. Note that $P \mapsto \Phi_P(v : \tau)$ is a linear form, i.e.

$$\Phi_{P+Q}(v : \tau) = \Phi_P(v : \tau) + \Phi_Q(v : \tau)$$
$$\Phi_{c \cdot P}(v : \tau) = c\Phi_P(v : \tau)$$

These properties are very useful, as they will allow us to directly manipulate annotations with $+$ and $\cdot$, knowing that the outputs of their corresponding potential functions will add and scale accordingly.

Additionally, we define several other useful constructs that operate on annotations.

- $\leq$: For $P, Q \in \mathcal{A}(\tau)$, we define preorder $P \leq Q$ to be $p_i \leq q_i$ for all $i \in \mathcal{I}(\tau)$.
- Product types: In the multivariate System GKH, we have the isomorphisms $\mathcal{I}(\tau_1 \times \tau_2) \cong \mathcal{I}(\tau_1) \times \mathcal{I}(\tau_2)$ and $\mathcal{A}(\tau_1 \times \tau_2) \cong \mathcal{A}(\tau_1) \otimes \mathcal{A}(\tau_2)$, motivating the helper function $\mathsf{pair} : \mathcal{A}(\tau_1) \times \mathcal{A}(\tau_2) \to \mathcal{A}(\tau_1 \times \tau_2)$. In univariate AARA, due to us not considering mixed potentials, the isomorphism is different, and essentially closer to that of sum types. Thus for product annotations, we will instead have the following helper function that actuates this isomorphism

  $$\mathsf{prj}_\ell : \mathcal{A}(\langle \ell : \tau_\ell \rangle_{\ell \in L}) \to \mathcal{A}(\tau_\ell)$$

  Intuitively, given the annotation of a product type, prj_ℓ extracts the sub-annotation for projection ℓ. It preserves potential as follows

  $$\Phi_P(\langle \ell : x_\ell \rangle_{\ell \in L} : \langle \ell : \tau_\ell \rangle_{\ell \in L}) = \sum_{\ell \in L} \Phi_{\mathsf{prj}_\ell(P)}(x_\ell : \tau_\ell)$$

- Sum types: Recall that we have added extra constant potential indices to each injection, meaning the inl and inr helpers will not be sufficient. Specifically, sum annotations will have the helper functions

  $$\mathsf{inj}_\ell^{-1} : \mathcal{A}([\ell : \tau_\ell]_{\ell \in L}) \to \mathcal{A}(\tau_\ell) \qquad \mathsf{inj}_\ell^{-*} : \mathcal{A}([\ell : \tau_\ell]_{\ell \in L}) \to \mathbb{Q}_{\geq 0}$$

  Note that we name these functions as "inverses" because injections are typically understood to go from a summand to the sum type, while inj_ℓ^{-1} and inj_ℓ^{-*} operate in the reverse direction. Intuitively, given the annotation of a sum type, inj_ℓ^{-1} extracts the sub-annotation for injection ℓ, and inj_ℓ^{-*} extracts our additional constant potential attached to ℓ. It preserves potential as follows

  $$\Phi_P(k \cdot x : [\ell : \tau_\ell]_{\ell \in L}) = \Phi_{\mathsf{inj}_k^{-1}(P)}(x : \tau_k) + \mathsf{inj}_k^{-*}(P)$$

- Recursive types: Recursive type annotations can be operated on by the shift operator, which corresponds to how potential is moved when unfolding the recursive type. The additive shift operator remains essentially the same as in System GKH, sans the constant indices.

  $$\triangleleft : \mathcal{A}(\mu t.\ \tau) \to \mathcal{A}([\mu t.\ \tau/t]\tau)$$

  The shift operator is the linear map corresponding to the function from basis elements $\mathcal{I}(\mu t.\ \tau) \to \mathcal{A}([\mu t.\ \tau/t]\tau)$ defined as $\triangleleft\ (\mathsf{fold} \cdot i) = i + \mathcal{M}\ \{t.\tau\}\ (\mathsf{fold} \cdot i)$. Intuitively, when unfolding a recursive type $\mu t.\ \tau$, the coefficient at some index $\mathsf{fold} \cdot i$ needs to be placed not only at index i in the output, but also copied to indices for all recursive occurrences of the type variable t in τ. For example, consider integer lists of type $\mu t.\ [\mathsf{nil} : \langle\rangle\ , \mathsf{cons} : \langle \mathsf{head} : \mathsf{int}, \mathsf{tail} : t \rangle]$, and note that the unfolded type is the sum type

  $$[\mathsf{nil} : \langle\rangle\ , \mathsf{cons} : \langle \mathsf{head} : \mathsf{int}, \mathsf{tail} : \mu t.\ [\mathsf{nil} : \langle\rangle\ , \mathsf{cons} : \langle \mathsf{head} : \mathsf{int}, \mathsf{tail} : t \rangle] \rangle]$$

  Now, consider index $\mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}}$, whose coefficient is the constant potential carried by each cons injection, or each element of the list. When unfolding the list, the shift operator ensures that said coefficient is copied to both $\mathsf{inj}^*_{\mathsf{cons}}$ and $\mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{tail}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}}$, which are respectively correspond to the constant potential of the *current* cons cell (if it is one), and the potential on each cons cell of the tail of the list.

  It preserves potential as follows

  $$\Phi_P(\mathbf{fold}\ v : \mu t.\ \tau) = \Phi_{\triangleleft\ P}(v : [\mu t.\ \tau/t]\tau)$$

- Sharing: Since we need to be able to use a value multiple times, we need to be able to split its potential across multiple uses. This is achieved by the sharing operator, which has the same specification in our univariate AARA as it does in System GKH.

  $$\curlyvee\colon \mathcal{A}(\tau) \times \mathcal{A}(\tau) \to \mathcal{A}(\tau)$$

  It does however have a much simpler definition. The multivariate nature of System GKH requires a carefully defined sharing operator for various pairs of indices to correctly handle potentials that are mixed multiplicatively. In the univariate setting, sharing two annotations simply shares the coefficients for each index disjointly, meaning that it coincides with the $+$ operator!

  $$P \curlyvee Q = P + Q$$

  It preserves potential as follows

  $$\Phi_P(v : \tau) + \Phi_Q(v : \tau) = \Phi_{P \curlyvee Q}(v : \tau)$$

  Notably, this differs from theorem 2 in Grosen et al's work, which states $\Phi_P(v : \tau) \cdot \Phi_Q(v : \tau) = \Phi_{P \curlyvee Q}(v : \tau)$, with a $\cdot$ instead of $+$. Once again, the difference arises because of the absence of multivariate mixed potential interactions during sharing.

As the core mechanism for assigning potential to typed values, annotations and their helper functions form the actual backbone of resource analysis, to be discussed in chapter 8.

Chapter 7

Resource Polynomial Implementation

In this chapter, we present the implementation of univariate AARA resource polynomials from chapter 6.

7.1 Base Polynomial Indices

The implementation of base polynomial indices follows quite directly from the definition of index set $\mathcal{I}(\tau)$ in figure 6.1 — `type t` is a recursive datatype representing the index variants.

```
type t =
| Iprj of Label.t * t
| Iinj of Label.t * t
| Iinjc of Label.t
| Ifold of t
```

Here is its associated interface:

```
module type S = sig
  type t
  include Comparable.S with type t := t

  val generate : max_deg:int -> Typ.t -> Set.t
  val degree : t -> int
  val to_string : t -> string
end
```

Listing 7.1: Base Polynomial Index Interface

The key difference is that while recursive types have infinite indices in the AARA theory, the implementation cannot have infinite sets. Thus, when generating $\mathcal{I}(\tau)$, `generate` takes in an argument `max_deg` to limit the number of fold's that can appear in an index. `degree` on the other hand returns the number fold's that appear in an index.

The notion of an index's degree being the number of fold's is somewhat informal. Recall from chapter 6 that for lists, the index

$$\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{tail}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}}$$

with 2 folds in it corresponds to the quadratic potential, or ordered pairs of list elements. The degree does correspond quite directly with the number of fold's. On the other hand, the notion breaks down somewhat on more complex data structures. Taking our rose tree example, recall that the index

$$\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{Rose}} \cdot \mathsf{prj}_{\mathsf{kids}} \cdot \mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{head}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{Rose}}$$

with 3 folds in it corresponds to ordered pairs of nodes and their descendants. The number of such ordered pairs is no longer an obvious polynomial with a degree — there are $\sum_{v \in T} \mathsf{depth}(v)$ such pairs in a rose tree T. This highlights the ability of our AARA to represent potential on complex, combinatorial structures. Anyway, for the purposes of ensuring a finite index set during implementation, our notion of a degree suffices.

7.2 Resource Annotations

From an implementation standpoint, a resource annotation for a RaML IR type in $\mathcal{A}(\tau)$ is simply a map from indices in $\mathcal{I}(\tau)$ to rationals in $\mathbb{Q}_{\geq 0}$. While the most obvious implementation would be an OCaml `Map.t` from indices to LP variables, we instead opted to use functions from indices to coefficients. Not only does this take advantage of lazy computation, but it actually makes the implementation more elegant. Furthermore, instead of using LP variables directly, we leverage the `SolverUtil.t` introduced in listing 5.3 that can represent a linear combination of LP variables along with a constant offset. This feature ends up being quite important in the implementation of resource annotations. Finally, having just a function from `Index.t` to `SolverUtil.t` would be insufficient, due to the infinite domain, so annotations also need to store the maximum degree for which they truly contain indices for. Thus, the underlying type for resource annotations is

```
type t = { map : Index.t -> SolverUtil.t; max_deg : int }
```

Here is its associated interface:

```
module type S = sig
  module Index : Index.S
  module SolverUtil : Solver_util.S

  type t
  type t' = { tau : Typ.t; annot : t }

  (* general operations *)
  val generate : max_deg:int -> ?zeroed:bool -> Typ.t -> t'
  val ( <= ) : t' -> t' -> unit
  val ( >= ) : t' -> t' -> unit
  val ( + ) : t' -> t' -> t'

  (* type specific operations *)
  val inj_inv : Label.t -> t' -> t' * SolverUtil.t
  val prj : Label.t -> t' -> t'
  val shift : t' -> t'

  (* others *)
  val minimize : t' -> unit
  val get_value : t' -> float Index.Map.t
  val to_string : t' -> string
end
```

Listing 7.2: Resource Annotation Interface

Note how all the interface functions actually interact with `type t' = { tau : Typ.t; annot : t }` instead of just `t`. `t'` packs the associated IR type τ of the annotation, which is helpful for error checking, as well as necessary for some of the interface functions. This extra information is exposed to the users of the interface for convenience later on in chapter 9, implementation of resource analysis.

First, let's look at the general operations.

- `generate ~max_deg tau` generates a fresh annotation for RaML IR type τ, allowing indices up to the specified maximum degree. It actually creates an underlying (non-function) map between indices up to the maximum degree and fresh `SolverUtil.t`'s using `SolverUtil.generate`. The `map` function it creates looks up this underlying map, and returns 0 if no entry is found. This is because for degrees greater than the specified maximum, we assume the coefficients are all 0.
- Per the $+$ operator from the annotation semimodule definition, `P + Q` is implemented via the pointwise addition of their coefficients across all indices. The implementation is quite concise and lazy thanks to `map` being a function. Letting P's map be `map1` and Q's map be `map2`, we can easily construct the map for $P + Q$ as

  ```
  let map i = SolverUtil.(map1 i + map2 i)
  ```

- `P <= Q` operator adds the necessary linear constraints (via `SolverUtil.( <= )`) to enforce that each coefficient in P is less than or equal to its corresponding coefficient in Q. The main catch here is that we need to actually obtain the coefficients by running the lazy map. To do this, we simply generate the necessary subset of indices by calling `Index.generate` with the maximum `max_deg` of the two inputs, and call the each of the two input `map` functions across the generated indices. This technique of running the lazy map will be necessary for `minimize`, `get_value`, and `to_string` as well. For clarity, here is its implementation

  ```
  let ( <= ) { tau; annot = { map = map1; max_deg = md1 } }
             { annot = { map = map2; max_deg = md2 }; _ } =
    let ids = Index.generate ~max_deg:(max md1 md2) tau1 in
    Set.iter ids ~f:(fun i -> SolverUtil.(map1 i <= map2 i))
  ```

  Listing 7.3: Annotation $\leq$ Implementation

Next, we have the type specific operations. These correspond to the helper functions defined in section 6.3 for sum, product, and recursive types. It is here that the decision to make `map` a function really shines.

- `inj_inv l P` returns `(Q, c)` where $Q = \mathsf{inj}_\ell^{-1}(P)$ and $c = \mathsf{inj}_\ell^{-*}(P)$. Letting P's map function be `smap`, the constant potential attached to injection ℓ can be accessed by calling `smap (Iinjc l)`, while the sub-annotation Q attached to the injection can have its map function defined as follows

  ```
  let map i = smap (Iinj (label, i))
  ```

- `prj l P` returns `Q` where $Q = \mathsf{prj}_k(P)$. Q's map function can be defined in a similar manner as in `inj_inv`.
- `shift P` returns `Q` where $Q = \triangleleft P$. Its implementation is a lot more complicated and involves implementing the recursive occurrence helper function $\mathcal{M}\{t.\tau\}(i)$.

Note that we do not implement the sharing operator $\curlyvee$ explicitly. As explained in chapter 6, in the univariate AARA setting, sharing $P \curlyvee Q = P + Q$, so the `( + )` operator will suffice.

Finally, we address the functions that actually concern solving the linear program.

- `minimize` **`P`** actually conducts LP solving to try to minimize the annotation. This is done after the resource analysis type inference pass (chapter 9) has generated the necessary linear constraints concerning the coefficients of the annotation. We minimize the coefficients of the annotation, going in order from coefficients for the largest degree index to those for the smallest degree index. If we think of an actual polynomial, such as $ax^2 + bx + c$, we would want to minimize it by first minimizing a, then b, then c, since larger degree terms have more sway on the cost.

  The incremental solve process involves invoking **`SolverUtil`**`.minimize` repeatedly, which in addition to obtaining the minimum coefficient value, also adds a constraint enforcing said minimum (see 5.3), so that the next call to **`SolverUtil`**`.minimize` for a different coefficient does not mess up the coefficient for the previous higher degree when LP solving.

- `get_value` **`P`** simply outputs an actual map from indices to floats. It should be called *after* calling `minimize`, so that there are concrete coefficients for each index that are the result of minimizing the annotation.

Chapter 8

Resource Analysis

In this chapter, we present the actual resource analysis algorithm of univariate AARA, which arises from its type system.

8.1 Resource Typing Judgement

Resource typing judgements in univariate AARA are of the form

$$\Gamma; p \vdash_c e : \langle \tau, P \rangle ; \Delta; q$$

which can be read as "in the annotated context Γ with p constant potential, under cost model c, expression e has annotated type $\langle \tau, P \rangle$ with Δ leftover resources." The rules for this judgement are presented later in figure 8.1. We first explain some of the constructs in the judgement, as well as differences with System GKH [2].

- Annotated Types and Contexts: An annotated type is a type-annotation pair $\langle \tau, P \rangle$, where $P \in \mathcal{A}(\tau)$. Γ and Δ in the judgement are both annotated contexts. which are mappings of variables to annotated types. Thus, for each variable in the context, an annotated context indicates not just its type, but the resources it carries.

  This notably differs from System GKH's formulation, which completely disentangles annotations from the typing context by creating an annotation P for the entire context as if it were a product type. System GKH's disentanglement is necessary for multivariate AARA, where potential can be mixed between different variables of the context, just as they can for different entries of a product. On the other side of the same coin, the potential across the variables of a context are disjoint in univariate AARA, so we can get away with leaving the annotations in. This also means that we do not need System GKH's pseudovariable $\circ$ to mark the potential carried by the output, as it is simply the annotation P from $e : \langle \tau, P \rangle$ in the judgement above.

  Note that these annotated types are completely different from annotated types in older versions of RaML, which are named and notated similarly. Whereas the type τ in our annotated type is a "vanilla" type that can stand alone, τ in older versions of RaML contains and is mutually recursive with annotated types.

- Constant Potential: As we have alluded to already, our choice of indices leaves out as many constant potentials as possible, adding the unavoidable ones for sum types. This means the constant potential function needs to exist somewhere else, passing the buck all the way to the typing judgement, where lowercase $p, q \in \mathbb{Q}_{\geq 0}$ respectively denote the input and remainder constant potentials. System GKH does not have this construct due to constant indices existing for all types in their annotations.

- Remainder Contexts: In our typing judgement, Δ is the remainder (annotated) context, and q is the remainder constant potential. For each mapping $x : \langle \tau, P \rangle$ in Δ, P represents the leftover resources on x after evaluating

e. Strictly speaking, it is only necessary to store the annotations in Δ, and the types can be omitted. However, storing them helps make both the typing rules and implementation cleaner. Also, note that in the typing judgement, $\mathrm{dom}\,\Gamma = \mathrm{dom}\,\Delta$ must hold, since we track resources leftover on all variables.

Existing versions of RaML do not use remainder contexts, instead requiring an explicit share construct in their programming languages to track when variables are used multiple times and need to have their potentials split. Not only did a cumbersome translation pass to litter share throughout a user program, they could not track non-monotone resources, such as memory. Remainder contexts elide the explicit share construct, and allow us to track non-monotone resources properly. System GKH's type system also uses remainder contexts in a conceptually identical fashion as us, differing only where we use annotated types and a separate constant potential.

- Cost Models: Just like in System GKH, we have the two models "cost-paid" (cp) and "cost-free" (cf). The "cost-paid" model corresponds to an effectful execution that actually measures the ticks in an expression, while the "cost-free" model corresponds to a pure execution does not, and merely "moves" existing potential around as dictated by the expression. Unlike in multivariate AARA, this cost model distinction is not necessary for the theory of univariate AARA. We nevertheless retain it because it will be useful in resource polymorphic recursion.

8.2 Annotated Function Types

Recall from chapter 2 that in both the theory and implementation of the RaML IR, function types have the straightforward form $\tau_1 \to \tau_2$. During the actual resource analysis, however, the reality is quite a bit more complicated, as we need to track how potential flows through a function. To this end, we will "cheat" a little and imitate System GKH, replacing the "vanilla" function types with

$$\langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle$$

where $\Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \subseteq (\mathcal{A}(\tau_1) \times \mathbb{Q}_{\geq 0}) \times (\mathcal{A}(\tau_2) \times \mathcal{A}(\tau_1) \times \mathbb{Q}_{\geq 0})$. Θ_{cp} and Θ_{cf} are families of resource specifications. A specification of the form

$$((P, p), (R, Q, q)) \in \Theta_c$$

states that the function should be called on an argument value v carrying potential $\Phi_P(v : \tau_1)$ and also be granted p constant potential; once the function executes under cost model c, it will return output value w carrying potential $\Phi_R(w : \tau_2)$, as well as having $\Phi_Q(v : \tau_1)$ potential leftover for the argument v, and q constant potential leftover. Note that under cost model cf, since no ticks are considered, $\Phi_P(v : \tau_1) + p = \Phi_R(w : \tau_2) + \Phi_Q(v : \tau_1) + q$.

A family, or set, of specifications is needed, to support resource polymorphic recursion. Initially, one may be tempted to store only one, "least" specification, representing the minimum resources that the function requires. However, as Hoffmann et al [5] discovered, this is insufficient for the analysis of certain recursive functions that call themselves more than once.

Another matter of note is that functions may only use constant potential and the potential carried by their argument. Functional programming paradigms encourage functions that use variables available in their current closure/context, but we forbid functions from consuming potential carried by such variables. If a function could consume potential from variables in its closure, our analysis would need to compute the number of times that it gets called, which is impossible to do statically (and could loop forever dynamically). Of course, functions are still allowed to use variables in their closure in a cost-free manner.

$$\frac{p \geq q}{\Gamma; p \vdash_{\mathsf{cp}} \mathsf{tick}\,[q] : \langle \langle\rangle, \varnothing \rangle ; \Gamma; p - q}\ \textsc{R:TICK}_{\mathsf{p}} \qquad \frac{}{\Gamma; p \vdash_{\mathsf{cf}} \mathsf{tick}\,[q] : \langle \langle\rangle, \varnothing \rangle ; \Gamma; p}\ \textsc{R:TICK}_{\mathsf{f}}$$

$$\frac{P \geq Q \curlyvee R}{\Gamma, x : \langle \tau, P \rangle ; p \vdash_c x : \langle \tau, Q \rangle ; \Gamma, x : \langle \tau, R \rangle ; p}\ \textsc{R:VAR}$$

$$\frac{\Gamma_1; p_1 \vdash_c e_1 : \langle \tau', R \rangle ; \Gamma_2; p_2 \quad \Gamma_2, x : \langle \tau', R \rangle ; p_2 \vdash_c e_2 : \langle \tau, Q \rangle ; \Gamma_3, x : \_; p_3}{\Gamma_1; p \vdash_c \mathsf{let}\,(e_1, x.\ e_2) : \langle \tau, Q \rangle ; \Gamma_3; p_3}\ \textsc{R:LET}$$

$$\frac{k \in L \quad P \geq \mathsf{inj}_k^{-1}(Q) \curlyvee R \quad p \geq \mathsf{inj}_k^{-*}(Q) + q}{\Gamma, x : \langle \tau_k, P \rangle ; p \vdash_c \mathsf{inj}\,[k]\,\left\{[\ell : \tau_\ell]_{\ell \in L}\right\}(x) : \left\langle [\ell : \tau_\ell]_{\ell \in L}, Q \right\rangle ; \Gamma, x : \langle \tau_k, R \rangle ; q}\ \textsc{R:INJ}$$

$$\frac{\begin{array}{c}(\forall \ell \in L) \quad \begin{array}{c}\mathsf{inj}_\ell^{-1}(P) \geq \mathsf{inj}_\ell^{-1}(R_\ell) \curlyvee S_\ell \qquad \mathsf{inj}_\ell^{-*}(P) + p \geq \mathsf{inj}_\ell^{-*}(R_\ell) + r_\ell \\ \Gamma, x : \left\langle [\ell : \tau_\ell]_{\ell \in L}, R_\ell \right\rangle, x_\ell : \langle \tau_\ell, S_\ell \rangle ; r_\ell \vdash_c e_\ell : \langle \tau, Q_\ell \rangle ; \Delta_\ell, x : \left\langle [\ell : \tau_\ell]_{\ell \in L}, T_\ell \right\rangle, x_\ell : \langle \tau_\ell, U_\ell \rangle ; s_\ell \\ \mathsf{inj}_\ell^{-1}(T_\ell) \curlyvee U_\ell \geq \mathsf{inj}_\ell^{-1}(R) \qquad \mathsf{inj}_\ell^{-*}(T_\ell) + s_\ell \geq \mathsf{inj}_\ell^{-*}(R) + q \\ \forall y \in \mathrm{dom}\,\Gamma.\ \Delta_\ell(y).P \geq \Delta(y).P \qquad Q_\ell \geq Q\end{array}\end{array}}{\Gamma, x : \left\langle [\ell : \tau_\ell]_{\ell \in L}, P \right\rangle ; p \vdash_c \mathsf{casesum}\,\{\tau\}\left(x, \{\ell : x_\ell.\ e_\ell\}_{\ell \in L}\right) : \langle \tau, Q \rangle ; \Delta, x : \left\langle [\ell : \tau_\ell]_{\ell \in L}, R \right\rangle ; q}\ \textsc{R:CASESUM}$$

$$\frac{\forall i \in [n].\ P_i \geq \mathsf{prj}_{\ell_i}(Q) \curlyvee R_i}{\Gamma, x_1 : \langle \tau_1, P_1 \rangle, ..., x_n : \langle \tau_n, P_n \rangle ; p \vdash_c \mathsf{tuple}\left(\{\ell_i : x_i\}_{i \in [n]}\right) : \left\langle \langle \ell_i : \tau_i \rangle_{i \in [n]}, Q \right\rangle ; \Gamma, x_1 : \langle \tau_1, R_1 \rangle, ..., x_n : \langle \tau_n, R_n \rangle ; p}\ \textsc{R:PROD}$$

$$\frac{\begin{array}{c}\forall i \in [n].\ \mathsf{prj}_{\ell_i}(P) \geq \mathsf{prj}_{\ell_i}(S) \curlyvee S_i \\ \Gamma, x : \left\langle \langle \ell_i : \tau_i \rangle_{i \in [n]}, S \right\rangle, x_1 : \langle \tau_1, S_1 \rangle, ..., x_n : \langle \tau_n, S_n \rangle ; p \\ \vdash_c e : \langle \tau, Q \rangle ; \Delta, x : \left\langle \langle \ell_i : \tau_i \rangle_{i \in [n]}, T \right\rangle, x_1 : \langle \tau_1, T_1 \rangle, ..., x_n : \langle \tau_n, T_n \rangle ; q \\ \forall i \in [n].\ \mathsf{prj}_{\ell_i}(T) \curlyvee T_i \geq \mathsf{prj}_{\ell_i}(R)\end{array}}{\Gamma, x : \left\langle \langle \ell_i : \tau_i \rangle_{i \in [n]}, P \right\rangle ; p \vdash_c \mathsf{caseprod}\left(x, \{\ell_i : x_i\}_{i \in [n]}\,.\ e\right) : \langle \tau, Q \rangle ; \Delta, x : \left\langle \langle \ell_i : \tau_i \rangle_{i \in [n]}, R \right\rangle ; q}\ \textsc{R:CASEPROD}$$

$$\frac{P \geq \triangleleft Q \curlyvee R}{\Gamma, x : \langle [\mu t.\ \tau/t]\tau, P \rangle ; p \vdash_c \mathsf{fold}\,\{t.\tau\}\,(x) : \langle \mu t.\ \tau, Q \rangle ; \Gamma, x : \langle [\mu t.\ \tau/t]\tau, R \rangle ; p}\ \textsc{R:FOLD}$$

$$\frac{\begin{array}{c}\triangleleft P \geq \triangleleft S \curlyvee T \\ \Gamma, x : \langle \mu t.\ \tau', S \rangle, y : \langle [\mu t.\ \tau'/t]\tau', T \rangle ; p \vdash_c e : \langle \tau, Q \rangle ; \Delta, x : \langle \mu t.\ \tau', U \rangle, y : \langle [\mu t.\ \tau'/t]\tau', V \rangle ; q \\ \triangleleft U \curlyvee V \geq \triangleleft R\end{array}}{\Gamma, x : \langle \mu t.\ \tau', P \rangle ; p \vdash_c \mathsf{casefold}\,(x, y.e) : \langle \tau, Q \rangle ; \Delta, x : \langle \mu t.\ \tau', R \rangle ; q}\ \textsc{R:CASEFOLD}$$

$$\frac{\begin{array}{c}\exists \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}}.\ \forall c \in \{\mathsf{cp}, \mathsf{cf}\}\,.\ \forall ((P, p), (Q, R, q)) \in \Theta_c. \\ \overline{\Gamma}, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle, x : \langle \tau_1, P \rangle ; p \vdash_c e : \langle \tau_2, Q \rangle ; \overline{\Gamma}, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle, x : \langle \tau_1, R \rangle ; r \\ \Gamma, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle \vdash_c e' : \tau\end{array}}{\Gamma; p \vdash_c \mathsf{letfun}\,(\{(f : \tau_1 \to \tau_2).x.e\}, f.e') : \tau; \Gamma; p}\ \textsc{R:LETFUN}$$

$$\frac{((Q, p), (R, S, q)) \in \Theta_c}{\Gamma, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle, x : \langle \tau_1, Q \rangle ; p \vdash_c \mathsf{app}\,(f, x) : \langle \tau_2, R \rangle ; \Gamma, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle, x : \langle \tau_1, S \rangle ; q}\ \textsc{R:APP}$$

Figure 8.1: Resource Typing Inference Rules

8.3 Resource Typing Rules

We finally arrive at the resource typing rules, the core of univariate AARA resource analysis. In System GKH, as well as most previous iterations of RaML, the structural rules are separate from the syntactic rules. However, to better motivate the implementation, our rules, presented in figure 8.1, will only contain syntactic, or algorithmic rules. We achieve this by subsuming the structural rules into certain syntactic rules when necessary.

We now examine some of the interesting rules:

- R:TICK$_\mathsf{p}$, R:TICK$_\mathsf{f}$: In the cost-paid setting, $\mathsf{tick}\,[q]$ consumes q constant potential. Because RaML forbids negative potentials, we ensure that the remainder potential $p - q$ is non-negative. In the cost-free setting, it does nothing. The empty product, or unit, has no indices and thus an empty annotation, which we denote as $\varnothing$.

- R:LET: This rule analyzes e_1, then uses the remainder context from this to analyze e_2. The only quirk is that the remainder potential on x when analyzing e_2 cannot be recovered, as x is a bound variable in this expression.

- R:INJ: The resources on x are shared between the remainder resources on x and injection k of the output. The constant potential p is shared between the remainder q and the constant potential for injection k of the output. Note that the coefficients corresponding to the other injections of the output do not have constraints enforced on them.

- R:CASESUM: Casing on sums is one of the messiest rules. We need to conduct an analysis for each branch ℓ of the case consisting of numerous steps. First, the resources on the case argument x need to be shared between x_ℓ and injection ℓ of x going into the input context for analyzing e_ℓ. The same needs to be done for the constant potential carried by the injection.

  Once e_ℓ is analyzed, the remaining resources carried on injection ℓ of x shared with the remaining resources on x_ℓ need to cover the resources on injection ℓ of x in the final output remainder context. Again, the same needs to be done for the constant potential.

  Finally, because each branch of the case can consume different amounts of resources carried by the context, the annotations for the result as well as each variable in the context will differ across all branches. The final output remainder context needs to essentially lower bound all of them. The notation $\Delta(y).P$ refers to the annotation attached to variable y in annotated context Δ. These premises are usually avoided by separating structural rules from syntactic rules, but we opted to do otherwise.

- R:CASEFOLD: Previously, when discussing the RaML IR in chapter 2, we noted that our elimination form for recursive types differed from System GKH, which uses unfold instead of casefold. The significance of this choice is finally manifest in this resource typing rule.

  When analyzing the subexpression e, y, which is essentially x unfolded, receives an annotation T which arises from sharing the shifted input annotation of x, P, between y and x. This first shift and share is already present in System GKH's rule for unfold. Our contribution is the ability to recover remainder resources on y and x, and share them back into the final remainder resources for x, with the necessary shifts. To see how this happens, consider the following expression in a language with unfold:

  $$\mathbf{let}\ y = \mathbf{unfold}\ x\ \mathbf{in}\ e$$

  The old typing rule for unfold could share the resources on x between x and y when analyzing e. However, if e contains non-monotone resource usage, namely recovering/gaining resources, there would be no way to properly share these resources back to x from y. Our casefold construct entirely elides this issue by essentially limiting the scope of y.

- R:LETFUN: While the RaML IR can support mutually recursive functions, we have postponed supporting them for actual resource analysis. Thus, our presentation of the resource typing rule for letfun only contains a single function.

  $\overline{\Gamma}$ denotes setting all coefficients in each annotation of annotated context Γ to 0. We do this before typing the function body of f to prevent it from consuming potential from its closure, as previously explained. Then, given a reference f to itself and argument x carrying resources represented by annotation P, the function body e should have annotation R, and Q remainder annotation on the argument. Note that the function types themselves have annotations too, but they are explicitly written out to be $\varnothing$, as functions carry no potential.

  There are technically infinite premises because each specification in the specification families Θ_{cp} and Θ_{cf} need to be validated. Of course, the actual implementation of the resource analysis will not actually need to check infinite premises.

  Finally, once the specification families are verified, the typing for function f can be used to type the continuation expression e'.

- R:APP: Due to being in the univariate setting, our version of the rule is significantly simpler than System GKH's rule for the multivariate setting. A function application is valid as long as the specification of the argument and result resources is in the specification family Θ_c.

8.4 Soundness

The soundness theorem for univariate AARA on our specific IR is essentially identical to older presentations of the soundness theorem. It relates the the statically derived resource bounds with actual resource use per the cost semantics (see chapter 2).

Theorem 1. *Let* $\Gamma; p \vdash_{\mathsf{cp}} e : \langle \tau, P \rangle ; \Delta; q$ *and* $V : \Gamma$. *Define the following quantities:*

$$I := p + \sum_{(x:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(x) : \tau) \qquad \text{(initial potential)}$$

$$F := q + \Phi_P(v : \tau) + \sum_{(x:\langle \tau, Q\rangle)\in\Delta} \Phi_Q(V(x) : \tau) \qquad \text{(final potential)}$$

If $V \vdash e \Downarrow v \mid (r, r')$ *for some value* v *and resources* (r, r')*, then*

$$I \geq r \text{ and } I - F \geq r - r'$$

The intial potential is the sum of the potentials across all values which the variables of context Γ map to, as well as the constant potential p. The final is the sum of the remainder potential of the variables in the context, the potential carried by result value v, and the remainder constant potential q.

The first inequality of the theorem states that the initial potential can cover the high-water mark cost r. The second inequality states that the difference between the initial potential and the final potential can cover the net cost $r - r'$.

Proof. The soundness proof proceeds by rule induction on the evaluation judgement. Our choice to go with purely syntax-directed typing rules means we can cite the (strong) inversion lemma for the typing judgements. We present a few representative cases, including our new recursive type elimination form casefold.

- We start with a base case, tick's that consume resources:

$$\frac{q \geq 0}{V \vdash \mathsf{tick}\,[q] \Downarrow \langle\rangle \mid (q, 0)}\ \text{E:TICK}^+$$

By typing inversion, we know the tick expression is typed by the following rule:

$$\frac{p \geq q}{\Gamma; p \vdash_{\mathsf{cp}} \mathsf{tick}\,[q] : \langle \langle\rangle , \varnothing \rangle ; \Gamma; p - q}\ \text{R:TICK}_{\mathsf{p}}$$

We now define the initial potential I and final potential F:

$$I := p + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau)$$
$$F := (p - q) + \Phi_\varnothing(\langle\rangle : \langle\rangle) + \sum_{(z:\langle \tau, Q\rangle) \in \Gamma} \Phi_Q(V(z) : \tau)$$

To finish this case, we show that $I \geq q$ and $I - F \geq q - 0$:

$$\begin{aligned}
I &= p + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) && \text{(definition of } I\text{)} \\
&\geq p && \text{(all potential evaluations are nonnegative)} \\
&\geq q && \text{(premise of typing rule)} \\
I - F &= \left(p + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) \right) - \left((p - q) + \Phi_\varnothing(\langle\rangle : \langle\rangle) + \sum_{(z:\langle \tau, Q\rangle) \in \Gamma} \Phi_Q(V(z) : \tau) \right) \\
& && \text{(definitions of } I \text{ and } F\text{)} \\
&= q - \Phi_\varnothing(\langle\rangle : \langle\rangle) && \text{(simplify)} \\
&= q - 0 \geq q - 0 && \text{(unit carries 0 potential)}
\end{aligned}$$

- Next, we consider the case of fold expressions:

$$\frac{}{V, x \mapsto v \vdash \mathsf{fold}\,\{t.\tau\}\,(x) \Downarrow \mathbf{fold}\ v \mid (0, 0)}\ \text{E:FOLD}$$

By typing inversion, we know the fold expression is typed by the following rule:

$$\frac{P \geq \triangleleft\, Q \curlyvee R}{\Gamma, x : \langle [\mu t.\ \tau/t]\tau, P\rangle ; p \vdash_c \mathsf{fold}\,\{t.\tau\}\,(x) : \langle \mu t.\ \tau, Q\rangle ; \Gamma, x : \langle [\mu t.\ \tau/t]\tau, R\rangle ; p}\ \text{R:FOLD}$$

We now define the initial potential I and final potential F:

$$I := p + \Phi_P(v : [\mu t.\ \tau/t]\tau) + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau)$$
$$F := p + \Phi_Q(\mathbf{fold}\ v : \mu t.\ \tau) + \Phi_R(v : [\mu t.\ \tau/t]\tau) + \sum_{(z:\langle \tau, Q\rangle) \in \Gamma} \Phi_Q(V(z) : \tau)$$

$I \geq 0$ trivially because all potential evaluations as well as any constant potential p are nonnegative. To finish this case, we show that $I - F \geq 0 - 0$ via $I \geq F$:

$$\begin{aligned}
I &= p + \Phi_P(v : [\mu t.\ \tau/t]\tau) + \sum_{(z:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(z) : \tau) && \text{(definiition of } I\text{)} \\
&\geq p + \Phi_{\triangleleft\, Q \curlyvee R}(v : [\mu t.\ \tau/t]\tau) + \sum_{(z:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(z) : \tau) && \text{(premise of typing rule)} \\
&= p + \Phi_{\triangleleft\, Q}(v : [\mu t.\ \tau/t]\tau) + \Phi_R(v : [\mu t.\ \tau/t]\tau) + \sum_{(z:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(z) : \tau) \\
& && (\curlyvee \text{ definition and } \Phi \text{ properties}) \\
&= p + \Phi_Q(\mathbf{fold}\ v : \mu t.\ \tau) + \Phi_R(v : [\mu t.\ \tau/t]\tau) + \sum_{(z:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(z) : \tau) && (\triangleleft\, Q \text{ preserves potential}) \\
&= F && \text{(definition of } F\text{)}
\end{aligned}$$

- Finally, we tackle our new recursive type elimination form casefold:

$$\frac{V, x \mapsto \mathbf{fold}\ v', y \mapsto v' \vdash e \Downarrow v \mid (r, r')}{V, x \mapsto \mathbf{fold}\ v' \vdash \mathsf{casefold}\,(x, y.e) \Downarrow v \mid (r, r')}\ \text{E:CASEFOLD}$$

By typing inversion, we know the casefold expression is typed by the following rule:

$$\frac{\begin{array}{c} \triangleleft\, P \geq \triangleleft\, S \curlyvee T \\ \Gamma, x : \langle \mu t.\ \tau', S\rangle, y : \langle [\mu t.\ \tau'/t]\tau', T\rangle ; p \vdash_c e : \langle \tau, Q\rangle ; \Delta, x : \langle \mu t.\ \tau', U\rangle, y : \langle [\mu t.\ \tau'/t]\tau', V\rangle ; q \\ \triangleleft\, U \curlyvee V \geq \triangleleft\, R \end{array}}{\Gamma, x : \langle \mu t.\ \tau', P\rangle ; p \vdash_c \mathsf{casefold}\,(x, y.e) : \langle \tau, Q\rangle ; \Delta, x : \langle \mu t.\ \tau', R\rangle ; q}\ \text{R:CASEFOLD}$$

Now, taking the premise of the evaluation rule and the 2nd premise of the typing rule, we apply the induction hypothesis on subexpression e. This gives us the initial potential I' and final potential F' arising from the resource typing of subexpression e:

$$\begin{aligned}
I' &:= p + \Phi_S(\mathbf{fold}\ v' : \mu t.\ \tau') + \Phi_T(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(z) : \tau) \\
F' &:= q + \Phi_Q(v : \tau) + \Phi_U(\mathbf{fold}\ v' : \mu t.\ \tau') + \Phi_V(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, Q\rangle)\in\Delta} \Phi_Q(V(z) : \tau)
\end{aligned}$$

which satisfy $I' \geq r$ and $I' - F' \geq r - r'$. We now define the initial potential I and final potential F for the overall expression $\mathsf{casefold}\,(x, y.e)$:

$$\begin{aligned}
I &:= p + \Phi_P(\mathbf{fold}\ v' : \mu t.\ \tau') + \sum_{(z:\langle \tau, P\rangle)\in\Gamma} \Phi_P(V(z) : \tau) \\
F &:= q + \Phi_Q(v : \tau) + \Phi_R(\mathbf{fold}\ v' : \mu t.\ \tau') + \sum_{(z:\langle \tau, Q\rangle)\in\Delta} \Phi_Q(V(z) : \tau)
\end{aligned}$$

We show that $I \geq I'$:

$$
\begin{aligned}
I &= p + \Phi_P(\mathbf{fold}\ v' : \mu t.\ \tau') + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) && \text{(definition of } I\text{)} \\
&= p + \Phi_{\triangleleft\, P}(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) && (\triangleleft\, P \text{ preserves potential}) \\
&\geq p + \Phi_{\triangleleft\, S \curlyvee T}(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) && \text{(1st premise of typing rule)} \\
&= p + \Phi_{\triangleleft\, S}(v' : [\mu t.\ \tau'/t]\tau') + \Phi_T(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) \\
& && (\curlyvee \text{ definition and } \Phi \text{ properties}) \\
&= p + \Phi_S(\mathbf{fold}\ v' : \mu t.\ \tau') + \Phi_T(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, P\rangle) \in \Gamma} \Phi_P(V(z) : \tau) && (\triangleleft\, S \text{ preserves potential}) \\
&= I' && \text{(definition of } I'\text{)}
\end{aligned}
$$

as well as $F \leq F'$:

$$
\begin{aligned}
F &= q + \Phi_Q(v : \tau) + \Phi_R(\mathbf{fold}\ v' : \mu t.\ \tau') + \sum_{(z:\langle \tau, Q\rangle) \in \Delta} \Phi_Q(V(z) : \tau) && \text{(definition of } F\text{)} \\
&= q + \Phi_Q(v : \tau) + \Phi_{\triangleleft\, R}(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, Q\rangle) \in \Delta} \Phi_Q(V(z) : \tau) && (\triangleleft\, R \text{ preserves potential}) \\
&\leq q + \Phi_Q(v : \tau) + \Phi_{\triangleleft\, U \curlyvee V}(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, Q\rangle) \in \Delta} \Phi_Q(V(z) : \tau) && \text{(3rd premise of typing rule)} \\
&= q + \Phi_Q(v : \tau) + \Phi_{\triangleleft\, U}(v' : [\mu t.\ \tau'/t]\tau') + \Phi_V(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, Q\rangle) \in \Delta} \Phi_Q(V(z) : \tau) \\
& && (\curlyvee \text{ definition and } \Phi \text{ properties}) \\
&= q + \Phi_Q(v : \tau) + \Phi_U(\mathbf{fold}\ v' : \mu t.\ \tau') + \Phi_V(v' : [\mu t.\ \tau'/t]\tau') + \sum_{(z:\langle \tau, Q\rangle) \in \Delta} \Phi_Q(V(z) : \tau) \\
& && (\triangleleft\, U \text{ preserves potential}) \\
&= F' && \text{(definition of } F'\text{)}
\end{aligned}
$$

All that is left to finish this case is to show that $I \geq r$ and $I - F \geq r - r'$:

$$
\begin{aligned}
I &\geq I' && \text{(shown above)} \\
&\geq r && \text{(induction hypothesis, stated above)} \\
I - F &\geq I' - F && (I \geq I', \text{ shown above}) \\
&= I' - F' && (F \leq F', \text{ shown above}) \\
&\geq r - r' && \text{(induction hypothesis, stated above)}
\end{aligned}
$$

The remaining cases proceed similarly:

- The E:TICK$^-$ case is similar to the presented E:TICK$^+$ case.
- The introduction form cases follow the structure of the presented E:FOLD case.
- The elimination form cases follow the structure of the E:CASEFOLD case.

□

Chapter 9

Resource Analysis Implementation

In this chapter, we present the implementation of univariate AARA resource analysis described in chapter 8.

9.1 Restricting Higher-Order Functions

Before we get to core resource analysis algorithm, we need to address a major shortcoming of RaML: higher-order functions. While the AARA type theory does not explicitly forbid the application of typing rules to higher-order functions (hofs), it is unable to derive resource bounds for certain classes of hofs. Recall from figure 8.1 the premise regarding the resource typing of function bodies:

$$\overline{\Gamma}, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle, x : \langle \tau_1, P \rangle ; p \vdash_c e : \langle \tau_2, Q \rangle ; \overline{\Gamma}, f : \langle \langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle, \varnothing \rangle, x : \langle \tau_1, R \rangle ; r$$

Note how we zero out the potential carried by other variables in Γ when analyzing the function body, denoted by $\overline{\Gamma}$, as we do not want it to consume potential from its closure. This limitation breaks the analysis of curried functions that "wish" to carry potential on their first arguments. Curried functions with two arguments are actually just hofs, which upon receiving their first argument, return another function that takes the nominal second argument as its argument. The function typing rule above prevents this inner function from using potential on anything aside from its argument, the nominal second argument. This means it cannot use potential carried by the first argument! To see this in action, consider the following:

```
fun append_bad a b =
  case a of
    [] => b
  | x :: xs => (Raml.tick 1.0; x :: (append_bad xs b))
```

Listing 9.1: Curried (Bad) Append

If we ignore the fact that this `append_bad` is a curried hof, and just consider its intended behavior, it is clear that the input list `a` needs to carry 1 unit of potential on each of its elements, or cons injections, while input list `b` does not need to carry any. This is because the function ticks (consumes) 1 unit of potential for every recursive call, and the number of recursive calls is the length of list `a`.

Unfortunately, `append_bad` is in fact a curried function: `append_bad a = fn b => ...`. The inner function `fn b => ...` is not allowed to use potential from anything except `b`, so it will not be able to correctly analyze the function body that is written to require potential from `a`.

Thankfully, the "fix" is simple. Once we uncurry `append_bad` so that it takes a pair of lists and returns a list, the resulting function can be easily analyzed by RaML:

```
fun append (a, b) =
  case a of
    [] => b
  | x :: xs => (Raml.tick 1.0; x :: (append (xs, b)))
```

Listing 9.2: Uncurried (Working) Append

This problem has plagued AARA since its inception. Older versions of RaML have mitigated it somewhat by adding code transformations in the RaML pipeline that uncurry functions under the hood, which we plan to add to RaML 2 eventually. For now, we simply avoid analyzing functions whose result types contain other function types, and throw an error if we come across such a function.

To clarify, AARA is quite capable of analyzing hofs without this issue. For example, the list map function takes as one of its arguments a function to be applied to every element of its list argument. It can be analyzed by AARA without issue. To keep things simple though, we also avoid writing the list map function in its curried form; that is, it receives the input list and function as a pair.

9.2 Function Queriers

Function types in the RaML IR are of the form $\tau_1 \to \tau_2$, whereas the AARA resource typing rules requires they be annotated with resource specifications: $\langle \tau_1 \to \tau_2, \Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \rangle$. Unlike resource annotations which represent potential carried by positive types (non-functions), resource specifications for functions describe the behavior of a pending computation — the function in question. Recalling that $\Theta_{\mathsf{cp}}, \Theta_{\mathsf{cf}} \subseteq (\mathcal{A}(\tau_1) \times \mathbb{Q}_{\geq 0}) \times (\mathcal{A}(\tau_2) \times \mathcal{A}(\tau_1) \times \mathbb{Q}_{\geq 0})$. Θ_{cp}, we may at first be tempted to represent families of resource specifications as

```
(Annotation.t * SolverUtil.t) * (Annotation.t * Annotation.t * SolverUtil.t) Set.t
```

Unfortunately, the set of valid argument, result, and remainder annotations is typically infinite, so this method is infeasible. Even worse, we sometimes have 0 information on what concrete functions will actually be associated with the function type, so we would not know what go put in the set in such cases.

While there is not much we can do (for now) about the latter case, if a function type is associated with a function body, we do have more information about its family of resource specifications. Although the family can still be infinite, each resource specification in it will satisfy the same (up to α-equivalence) linear constraints relating the argument and result/remainder annotations. Specifically, these constraints will arise from the function body. Still, a fresh set of linear constraints needs to be generated each time a function of this type is actually applied. Thus, we represent families of resource specifications with the (OCaml) type `fn_query` defined as follows, where `fn_entry` is a single concrete resource specification:

```
type fn_entry = {
  arg_in : Annotation.t';
  const_in : Annotation.SolverUtil.t;
  arg_rem : Annotation.t';
  output : Annotation.t';
  const_out : Annotation.SolverUtil.t;
}

type fn_query = max_deg:int -> cost_free:bool -> fn_entry
```

Listing 9.3: (Incomplete) Function Querier Type

Every time we come across the function application of a (IR) function of an annotated function type, we can call its associated querier (`fn_query`) with the desired maximum degree and cost model. The querier should reproduce the

linear constraints relating the argument and result/remainder annotations by reanalyzing the function body. The core analysis algorithm can then work with the annotations of `fn_entry`, adding in further constraints relating `arg_in` to the concrete argument's annotation, and so on.

This `fn_query` is not quite complete though, as the caption of listing 9.3 suggests. While it is able to capture the behavior of IR functions whose argument and result types carry potential, it cannot deal with higher-order functions whose arguments and results themselves contain function types. For example, a querier for an IR list map function, $\mathsf{map}_{\mathsf{list}}$ will need information about the function f being mapped across the list, which the current `fn_query` type does not supply. This shortcoming will be partially addressed in the next section.

9.3 Function Annotations

How do these queriers fit into the implementation though? While we could have modified (or made a new version of) the RaML IR types which add a `fn_query` to all IR function types, we instead opted to create a new form of annotations: *function annotations*. A function annotation is simply how we mark the queriers for all the function types that appear within a type. We implement them using the familiar system of a map function (see implementation of resource annotations in chapter 7):

```
type t = FnIndex.t -> fn_query option
```

Function annotations map to `fn_query option` instead of just `fn_query` so that function types for which we know nothing about their behavior can be annotated with `None`.

The index set for function annotations of a specific type τ, $\mathcal{I}_f(\tau)$, are simpler than those for resource annotations, and serve only to mark where in the IR type AST the function type in question is. Here are the function annnotation indices:

$$\frac{k \in L \quad i \in \mathcal{I}_f(\tau_k)}{\mathsf{inj}_k \cdot i \in \mathcal{I}_f([\ell : \tau_\ell]_{\ell \in L})} \qquad \frac{k \in L \quad i \in \mathcal{I}_f(\tau_k)}{\mathsf{prj}_k \cdot i \in \mathcal{I}_f(\langle \ell : \tau_\ell \rangle_{\ell \in L})} \qquad \frac{i \in \mathcal{I}_f([\mu t.\ \tau/t]\tau)}{\mathsf{fold} \cdot i \in \mathcal{I}_f(\mu t.\ \tau)} \qquad \frac{}{\mathsf{arr} \in \mathcal{I}_f(\tau_1 \to \tau_2)}$$

Figure 9.1: Function Annotation Index Set Inference Rules

Every function annotation index "ends" in a arr. Note that nested function types, i.e. function types within the τ_1 and τ_2 of another function type $\tau_1 \to \tau_2$, cannot be referenced by these indices. This is an intentional limitation that would not be set had we modified the RaML IR types instead. As an example, consider the IR type $[\mathsf{Fn} : \tau_1 \to \tau_2, \mathsf{Nothing} : \langle\rangle]$. Its function annotation would be the mapping $\{\mathsf{inj}_{\mathsf{Fn}} \cdot \mathsf{arr} : q\}$. It contains one entry because the IR type only has one function annotation index, marking the position of the sub-type $\tau_1 \to \tau_2$ within the Fn injection of the sum type. q here denotes the querier.

Function annotations require a lot of unavoidable overhead, and even subtyping. Let's go back to the example $[\mathsf{Fn} : \tau_1 \to \tau_2, \mathsf{Nothing} : \langle\rangle]$. Consider some variable x of this type, and the expression

$$\textbf{match}\ x\ \textbf{with}\ [\mathsf{Fn} \cdot f \mapsto e_1 \mid \mathsf{Nothing} \cdot u \mapsto e_2]$$

When analyzing the subexpression e_1, we need to know the function annotation of the newly bound variable f. We can get this from taking the function annotation of x and extracting the sub-annotation corresponding to injection Fn, giving us $\{\mathsf{arr} : q\}$. In fact, things can get a lot worse: suppose that the two branches e_1 and e_2 are both functions. For simplicity, e_1 is a function that requires a_i constant potential to run, and leaves behind a_o constant potential, while function e_2 requires b_i input constant potential and leaves behind b_o constant potential. Then the function querier within the function annotation of this casesum expression will need to subtype both branches, producing a function type that requires c_i input and c_o output where $c_i \geq a_i, b_i$ and $c_o \leq a_o, b_o$.

We now circle back to the issue presented at the end of the last section, which is that function queriers are unable to handle higher-order functions. Recall the example that a querier for $\mathsf{map}_{\mathsf{list}}$ does not have access to the querier of the function f being mapped across the list, which the current `fn_query` type does not supply. However, with function annotations now defined, we can modify the `fn_query` type to also take in an extra argument: the function annotation of the (IR function's) argument. This results in a mutually recursive definition between function annotations and queriers as follows:

```
type fn_entry = { (* unchanged *) }

and fn_query = max_deg:int -> cost_free:bool -> t' -> fn_entry
and t = Index.t -> fn_query option (* function annotation *)
and t' = { tau : Typ.t; map : t }
```

Listing 9.4: Complete Function Querier and Annotation Type

Observe how `fn_query` has been modified to also take in a `t'` argument. Careful readers may wonder why the querier does not also return a function annotation `t'` — after all, the result type of a function may contain functions. This is because, as explained in section 9.1, we avoid analyzing functions whose result types contain other function types.

9.4 Resource Analysis Core

The main function of the core resource analysis algorithm is `inferCost`, which has the following interface:

```
type in_ctx = {
  a_ctx : Annotation.t' Var.Exp_var.Map.t;
  fn_ctx : FnAnnot.t' Var.Exp_var.Map.t;
  const : SolverUtil.t
}

type out_ctx = {
  a_ctx : Annotation.t' Var.Exp_var.Map.t;
  const : SolverUtil.t;
  a_out : Annotation.t';
  fn_out : FnAnnot.t';
}

val inferCost : max_deg:int -> cost_free:bool -> in_ctx -> Exp.t -> out_ctx
```

Listing 9.5: Resource Analysis Interface

Recall the form of the resource typing judgement $\Gamma; p \vdash_c e : \langle \tau, P \rangle ; \Delta; q$ from chapter 8. In `type in_ctx`, which is an input to `inferCost`, input annotated context Γ is `a_ctx` and input constant potential p is `const`. In `type out_ctx`, which is the result of `inferCost`, remainder annotated context Δ is `a_ctx`, and q is `const`, and $\langle \tau, P \rangle$ is `a_out`. The extra `fn_ctx` and `fn_out` arise to bookkeep function annotations.

`inferCost` cases on the provided IR expression, and generates the relevant linear constraints following the resource typing rules from figure 8.1. For example, recall the resource typing rule for variables:

$$\frac{P \geq Q \curlyvee R}{\Gamma, x : \langle \tau, P \rangle ; p \vdash_c x : \langle \tau, Q \rangle ; \Gamma, x : \langle \tau, R \rangle ; p} \text{ R:VAR}$$

The following is implementation of this rule:

```
match Exp.out exp with
| Evar x ->
    let a_in = inferCostVar a_ctx x in
    let a_out = Annotation.generate ~max_deg a_in.tau in
    let a_rem = Annotation.generate ~max_deg a_in.tau in
    Annotation.(a_in >= a_out + a_rem);
    let a_ctx = Map.set a_ctx ~key:x ~data:a_rem in
    { a_ctx; const; a_out; fn_out = ... }
...
```

Listing 9.6: Resource Analysis Variable Case

`a_in` is the resource annotation P of variable `x` in the input annotated context. The rule states that P needs to cover $Q \curlyvee R$, where Q is the annotation of the single-variable expression x, and R is the remainder annotation of x in the remainder context. To implement this, we generate fresh annotations `a_out` for Q and `a_rem` for R. Next, the key piece of code is `Annotation.(a_in >= a_out + a_rem)`, which adds linear constraints to the LP solver backend enforcing the $P \geq Q \curlyvee R$ constraint from the rule. Then, we construct the remainder context, which is just the input context but with `x` mapping to the remainder annotation `a_rem`, or R. Finally, we output the necessary fields: the remainder context, output annotated type, etc.

However, all `inferCost` directly does is generate the linear constraints that arise from typing an expression. What we are actually interested in for RaML is analyzing *functions*. Specifically, we would like to know the minimum resources we can supply via the argument and constant potential, and still cover the cost of all the ticks in a function. The following is the interface regarding function analysis:

```
type fn_concrete_in = {
  arg_in : float Annotation.Index.Map.t;
  const_in : float;
}

val analyzeFunction : fn_query -> fn_concrete_in
```

Listing 9.7: Function Analysis Interface

In `analyzeFunction`, we first call the `fn_query` associated with the function of interest to obtain an instance of the function's resource specification (of type `fn_entry`), which contains the argument input annotation `arg_in` and input constant potential `const_in`. We then call `Annotation.minimize` (from listing 7.2) to minimize `arg_in` (along with `SolverUtil.minimize` on `const_in`). Finally, `Annotation.get_value` gives us the mapping of indices to concrete float coefficients, allowing us to populate `fn_concrete_in`. These concrete resource annotations are the end of the RaML 2 pipeline. Examples of deriving concrete resource bounds on functions are the focus of the next chapter, chapter 10.

Chapter 10

Resource Analysis Examples

In this chapter, we examine how our RaML 2 implementation performs on various SML functions. For each function, we present the SML source code and the analysis results next to each other. A successful result will consist of an input annotation (and input constant potential) with concrete float coefficients, as well as the maximum polynomial degree required and number of constraints generated. We will also show an example that RaML is unable to analyze.

10.1 Identity with Tick

As a sanity check, we analyze the identity function with a tick.

```
fun id0 (x : int) = (Raml.tick 81.4; x)
```

```
max_deg -> 0
arg_in -> {}
const_in -> 81.40
```

Listing 10.1: Analysis of Identity with Tick

As expected, the analysis states that the function requires a minimum of 81.4 constant potential.

10.2 Quadratic List Identity

We start with `id1`, a linear time list identity function that recreates the list, while consuming 31.2 resources per cons-cell creation.

```
fun id1 l =
  case l of
    [] => []
  | x::xs => (Raml.tick 31.2; x::(id1 xs))
```

```
max_deg -> 1
arg_in -> {
  fold.in[::]*: 31.20,
  fold.in[nil]*: 0
}
const_in -> 0
```

Listing 10.2: Analysis of Linear List Identity

The analysis assigns a coefficient of 31.2 for the linear index `fold.in[::]*`, corresponding to `l` carrying

$$31.2\binom{n}{1}$$

potential when it has length n.

We now use `id1` as a building block for `id2`, a quadratic time list identity function.

```
fun id2 l =
  case l of
    [] => []
  | x::xs => (Raml.tick 411.0; x :: (id2
  ↪  (id1 xs)))
```

```
max_deg -> 2
arg_in -> {
  fold.in[::].pr[1].fold.in[::]*: 31.20,
  fold.in[::].pr[1].fold.in[nil]*: 0,
  fold.in[::]*: 411.00,
  fold.in[nil]*: 0
}
const_in -> 0
```

Listing 10.3: Analysis of Quadratic List Identity

This is a more interesting, albeit still trivial, function to analyze. The linear index `fold.in[::]*` is assigned 411 potential, which is directly consumed in the function body of `id2` every cons-cell creation. The more interesting index is `fold.in[::].pr[1].fold.in[::]*`. This is in fact the quadratic index of lists, depicted previously in chapter 6 as $\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{tail}} \cdot \mathsf{fold} \cdot \mathsf{inj}^*_{\mathsf{cons}}$, corresponding to ordered pairs of elements. Since each call to `id1` costs 31.2 linear potential, and `id1` is invoked once for every element of the input list, it is correct that the analysis deduces that there should be 31.2 quadratic potential. More concisely, the analysis computes that

$$31.2\binom{n}{2} + 411\binom{n}{1}$$

input potential is required on the input list `l` of length n. In addition to the correct analysis, this test case also shows RaML's ability to analyze interprocedural code, with call to `id1` made from within `id2`.

10.3 Resource Polymorphic Recursion

We now present an example that exercises RaML's ability to handle resource polymorphic recursion. Resource polymorphic recursion is when a function's recursive call within its body has a different resource specification than the resource specification of the body as a whole.

`id3` below calls `id2` from the previous section twice. The outer call can be typed easily without resource polymorphic recursion, but the inner call to `id2` will require resource polymorphic recursion to correctly obtain its resource specification.

```
fun id3 l = id2 (id2 l)
```

```
max_deg -> 2
arg_in -> {
  fold.in[::].pr[1].fold.in[::]*: 62.40,
  fold.in[::].pr[1].fold.in[nil]*: 0,
  fold.in[::]*: 822.00,
  fold.in[nil]*: 0
}
const_in -> 0
```

Listing 10.4: Analysis of Resource Polymorphic Quadratic List Identity

Without resource polymorphic recursion, RaML would have declared the LP infeasible. Since we did implement this feature, RaML correctly analyzes `id3`, assigning twice the potential to the input list as it did to `id2`, namely

$$62.4\binom{n}{2} + 822\binom{n}{1}$$

potential required where `l` has length n.

10.4 Quicksort

We now really step up the difficulty and analyze integer quicksort.

```
fun append (a, b) =
  case a of
    [] => b
  | x :: xs => (Raml.tick 4.11; x ::
  ↪  (append (xs, b)))

fun partition (p, xs) =
  case xs of
    [] => ([], [])
  | x :: xs =>
    let
      val (ls, gs) = partition (p, xs)
      val _ = Raml.tick 4.11
    in
      if x < p
      then (x :: ls, gs)
      else (ls, x :: gs)
    end


fun quicksort l =
  case l of
    [] => []
  | x :: xs =>
    let
      val (ls, gs) = partition (x, xs)
      val sls = quicksort ls
      val sgs = quicksort gs
    in
      append (sls, (x :: sgs))
    end
```

```
max_deg -> 2

append:
arg_in -> {
  pr[0].fold.in[::]*: 4.11,
  pr[0].fold.in[nil]*: 0,
  pr[1].fold.in[::]*: 0,
  pr[1].fold.in[nil]*: 0
}

partition:
arg_in -> {
  pr[1].fold.in[::]*: 4.11,
  pr[1].fold.in[nil]*: 0
}
const_in -> 0

quicksort:
arg_in -> {
  fold.in[::].pr[1].fold.in[::]*: 8.22,
  fold.in[::].pr[1].fold.in[nil]*: 0,
  fold.in[::]*: 0,
  fold.in[nil]*: 0
}
const_in -> 0
```

Listing 10.5: Analysis of Quicksort

Quicksort is built up using the helper functions `append` and `partition`, both of which tick 4.11 whenever they create a cons cell. RaML correctly determines that `append`'s first input list should carry 4.11 potential per element, while `partition`'s input list (which is the second projection, as the first is the integer pivot) should carry 4.11 potential per element as well.

Finally, it analyzes `quicksort`, which has a tricky recursion pattern that also intermixes with the calls to `append` and `partition`. Nevertheless, RaML correctly determines that the input list to `quicksort` should carry

$$8.22\binom{n}{2}$$

potential where it has length n. This covers the worst case where the partition function always places everything in list `sls`, requiring `append` to iterate over everything.

10.5 Append then Quicksort

Interestingly, quicksort also allows us to show an example that RaML is currently unable to analyze. While System GKH's multivariate AARA theory can analyze programs involving multiplicative resources, our current implementation of RaML is based on the more limited univariate AARA. Thus, it is unable to analyze `qsort2` below, which given two lists, appends them to each other and quicksorts them.

```
fun qsort2 (l1, l2) =
  quicksort (append (l1, l2))
```

```
LP infeasible
```

Listing 10.6: Analysis of Quicksorting Two Lists Appended

This is because quicksort is quadratic in cost in the length of the list being sorted, which is the sum of the lengths of the two input lists. Recalling that our previously defined quicksort costs $8.22\binom{n}{2}$ on a list of length n, a function that appends a list of length n_1 with a list of length n_2 then sorts them would cost

$$8.22\binom{n_1+n_2}{2} = 8.22\left(\binom{n_1}{2} + \binom{n_1}{1}\binom{n_2}{1} + \binom{n_2}{2}\right)$$

per Vandermonde's identity. The term $\binom{n_1}{1}\binom{n_2}{1}$ is precisely the sort of multiplicative resource that univariate AARA is unable to capture.

10.6 List Map

As explained in chapter 9, function annotations allow us to analyze higher-order functions like list `map`. To be clear, RaML does not produce a meaningful analysis on list `map` directly, as the function being mapped across the list is unknown. Instead, to get meaningful results, we create the additional function `map_id1` that maps `id1` from before across the list. Namely, the argument is a list of lists.

```
fun map (f, l) =
  case l of
    [] => []
  | x::xs => (f x) :: (map (f, xs))

fun map_id1 ll = map (id1, ll)
```

```
max_deg -> 2

map_id1:
arg_in -> {
  fold.in[::].pr[0].fold.in[::]*: 31.20,
  fold.in[::].pr[0].fold.in[nil]*: 0,
  fold.in[::].pr[1].fold.in[::]*: 0,
  fold.in[::].pr[1].fold.in[nil]*: 0,
  fold.in[::]*: 0,
  fold.in[nil]*: 0
}
const_in -> 0
```

Listing 10.7: Analysis of List Map

RaML is able to correctly determine that each of the inner lists should carry 31.2 potential on each of their elements, which lines up with the previous analysis of `id1` in listing 10.2.

The indices for linear potential on the inner list may appear to be rather similar to those of quadratic potential on the outer list; the key difference is the `pr[0]` instead of `pr[1]` in the indices, denoting that we are examining the head of the cons injection of the outer list, not the tail.

10.7 Rose Trees to Lists

While it is important to verify that our new RaML can properly analyze functions involving lists, its major advancement is its ability to analyze regular recursive types. Thus, in the following example, we will analyze `rose2list`, which flattens a rose tree into a list.

```
datatype 'a rose = Rose of 'a * 'a list

fun r2l (Rose (p, rs), l) =
  case rs of
    [] => (Raml.tick 4.11; p :: l)
  | x :: xs => r2l (Rose (p, xs), r2l (x,
  ↪  l))

fun rose2list r = r2l (r, [])
```

```
max_deg -> 1

rose2list:
arg_in -> {
  fold.in[Rose]*: 4.11
}
const_in -> -0.00
```

Listing 10.8: Analysis of Rose Trees to List

Note that we deliberately wrote `rose2list` as efficiently as possible by setting up the tail recursive `r2l` helper function. This helper function iterates over all nodes in the rose tree exactly once, and ticks 4.11 resources for each of them. RaML is able to analyze it and correctly conclude that `rose2list` requires the input rose tree to carry

$$4.11|T|$$

potential, where $|T|$ is the number of nodes in rose tree T.

10.8 Rose Trees to Lists — Slow

The previous example shows that RaML works on rose trees, but it was not very interesting, as it only dealt with linear potential. This was already achieved by Jost et al [8] well before Grosen et al. We now look at a more interesting example that arises from a more inefficient implementation of `rose2list`: `rose2list_slow`.

```
datatype 'a rose = Rose of 'a * 'a list

fun append (a, b) =
  case a of
    [] => b
  | x :: xs => (Raml.tick 4.11; x ::
  ↪  (append (xs, b)))

fun rose2list_slow (Rose (p, rs)) =
  case rs of
    [] => [p]
  | x :: xs => append (rose2list_slow x,
  ↪  rose2list_slow (Rose (p, xs)))
```

```
max_deg -> 4

rose2list_slow:
arg_in -> {
  fold.in[Rose].pr[1].fold.in[::]
              .pr[0].fold.in[Rose]*: 4.11,
  ...
}
const_in -> 0.00
```

Listing 10.9: Analysis of *Slow* Rose Trees to List

`rose2list_slow` is implemented in a more "natural" manner than the efficient tail recursive `r2l` shown previously. After making recursive calls on the tail list and the head child tree, `rose2list_slow` calls `append` to combine them, which costs 4.11 potential per element of the first input list. This significantly complicates the analysis of `rose2list_slow` — each node in the rose tree will end up contributing cost proportional to its depth in the tree!

Thankfully, our AARA has an expressive enough class of indices to express this potential. We have actually seen the index of interest above in the form

$$\mathsf{fold} \cdot \mathsf{inj}_{\mathsf{Rose}} \cdot \mathsf{prj}_{\mathsf{kids}} \cdot \mathsf{fold} \cdot \mathsf{inj}_{\mathsf{cons}} \cdot \mathsf{prj}_{\mathsf{head}} \cdot \mathsf{fold} \cdot \mathsf{inj}^{*}_{\mathsf{Rose}}$$

This index represents all ordered pairs consisting of a rose tree node and one of its descendant nodes, of which there are $\sum_{v \in T} \mathsf{depth}(v)$ pairs in a rose tree T. Thus, RaML has correctly deduced that the input rose tree should carry

$$4.11 \sum_{v \in T} \mathsf{depth}(v)$$

potential.

Chapter 11

Discussion

11.1 Future Work

11.1.1 Multivariate RaML

In this thesis, we present a univariate AARA type theory for a language with regular recursive types. On the other hand, System GKH [2], on which the theory in this thesis is heavily based on, presents a multivariate AARA theory. While our univariate AARA is unique, its expressive power is still a strict subset of multivariate AARA. The most obvious next step for RaML 2 is to implement type inference for multivariate AARA. Once it is implemented, many more programs can be analyzed, including listing 10.6, which cannot be analyzed by univariate AARA.

11.1.2 Exception Handling

Another direction in which RaML 2 can be improved is by adding support for exception handling. As mentioned earlier in the thesis already, the implementation of our RaML IR actually already contains expressions for throwing and handling exceptions. Additionally, our SML frontend can translate exceptions and their associated syntax in SML into their corresponding constructs in the RaML IR. The next step is clearly to incorporate and implement the theory from Guo's thesis [3] on adding exception handling to AARA. Once implemented, RaML 2 will be able to analyze programs with more unique control flow.

11.1.3 LP Backends

Our RaML 2 implementation currently uses the COIN-OR LP solver as its LP backend. While it largely suffices, we plan to add support for more LP backends in the future, notably Gurobi, as well as user customizable backends. Gurobi offers many more algorithms for solving linear programs, especially some specialized for flow-like LPs, which RaML linear constraints often are.

11.2 Conclusion

Automatic Amortized Resource Analysis (AARA) is a powerful technique that can infer resource bounds using a type system and the physicist's method of amortized analysis. This thesis presents a fresh implementation of AARA, Resource Aware ML (RaML) 2, which incorporates the latest advances in AARA theory and emphasizes modularity and maintainability. We showcase the various stages of RaML 2's pipeline all the way from its frontend to the outputting of concrete resource bounds.

Bibliography

[1] Matthew Fluet. MLton Compiler Overview. http://mlton.org/CompilerOverview, 2021. 4.3

[2] Jessie Grosen, David M. Kahn, and Jan Hoffmann. Automatic Amortized Resource Analysis with Regular Recursive Types. In *LICS*, pages 1–14, 2023. doi: 10.1109/LICS56636.2023.10175720. URL https://doi.org/10.1109/LICS56636.2023.10175720. 1.1, 2.1, 6, 6.3, 8.1, 11.1.1

[3] Yiyang Guo. Automatic Amortized Resource Analysis for Exception Handling. Master's thesis, Carnegie Mellon University, 2023. 11.1.2

[4] Robert Harper. *Practical Foundations for Programming Languages*, chapter 1.2. Cambridge University Press, 2nd edition, 2016. ISBN 1107150302. 3.1

[5] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010. doi: 10.1007/978-3-642-11957-6_16. URL https://doi.org/10.1007/978-3-642-11957-6_16. 1.1, 8.2

[6] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012. doi: 10.1007/978-3-642-31424-7_64. URL https://doi.org/10.1007/978-3-642-31424-7_64. 1.2

[7] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards Automatic Resource Bound Analysis for OCaml. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 359–373. ACM, 2017. doi: 10.1145/3009837.3009842. URL https://doi.org/10.1145/3009837.3009842. 1.2, 2.3

[8] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 223–236. ACM, 2010. doi: 10.1145/1706299.1706327. URL https://doi.org/10.1145/1706299.1706327. 10.8

[9] David M. Kahn and Jan Hoffmann. Automatic Amortized Resource Analysis with the Quantum Physicist's Method. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi: 10.1145/3473581. URL https://doi.org/10.1145/3473581. 1.2

[10] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6 (2):306–318, 1985. doi: 10.1137/0606031. URL https://doi.org/10.1137/0606031. 1.1