Compiling with Types

Greg Morrisett

December, 1995

CMU–CS–95–226

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
Robert Harper, Co-Chair
Jeannette Wing, Co-Chair
Peter Lee
Andrew Appel, Princeton University

This research was sponsored in part by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168. Support also was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency ([ARPA]) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the United States Government.

Keywords: compiling, types, Standard ML, unboxing, garbage collection, closure conversion.

Abstract

Compilers for monomorphic languages, such as C and Pascal, take advantage of types to determine data representations, alignment, calling conventions, and register selection. However, these languages lack important features including polymorphism, abstract datatypes, and garbage collection. In contrast, modern programming languages such as Standard ML (SML), provide all of these features, but existing implementations fail to take full advantage of types. The result is that performance of SML code is quite bad when compared to C.

In this thesis, I provide a general framework, called *type-directed compilation*, that allows compiler writers to take advantage of types at all stages in compilation. In the framework, types are used not only to determine efficient representations and calling conventions, but also to prove the correctness of the compiler. A key property of type-directed compilation is that all but the lowest levels of the compiler use *typed intermediate languages*. An advantage of this approach is that it provides a means for automatically checking the integrity of the resulting code.

An important contribution of this work is the development of a new, statically-typed intermediate language, called λ_i^{ML}. This language supports *dynamic type dispatch*, providing a means to select operations based on types at run time. I show how to use dynamic type dispatch to support polymorphism, ad-hoc operators, and garbage collection without having to box or tag values. This allows compilers for SML to take advantage of techniques used in C compilers, without sacrificing language features or separate compilation.

To demonstrate the applicability of my approach, I, along with others, have constructed a new compiler for SML called TIL that eliminates most restrictions on the representations of values. The code produced by TIL is roughly twice as fast as code produced by the SML/NJ compiler. This is due at least partially to the use of natural representations, but primarily to the conventional optimizer which manipulates typed, λ_i^{ML} code. TIL demonstrates that combining type-directed compilation with dynamic type dispatch yields a superior architecture for compilers of modern languages.

Acknowledgements

I owe a great deal of thanks to a great number of people. My wife Tanya (also known as Jack), went well beyond the reasonable call of duty in her help and support. Not only did she take care of day-to-day tasks (such as making dinner), but she also helped proofread this document.

This thesis and my work in general are direct reflections of Bob Harper's teaching and mentoring. Jeannette Wing also deserves a great deal of credit: Not only did she suffer through early drafts of this document, but she supported me in many other ways throughout my stay at CMU. Peter Lee, Andrew Appel, and Matthias Felleisen also deserve a great deal of credit for passing on their knowledge, wisdom, and experience. I am deeply honored to have studied under all of these people.

The TIL compiler wouldn't be here without David Tarditi, Perry Cheng, or Chris Stone. I truly enjoyed working with all of these guys and hope to do so again. Andrzej Filinski has been the ultimate officemate throughout my graduate career. Many other folks have been especially good to me over the years and deserve special recognition including Nick Barnes, Lars Birkedal, Sharon Burks, Prasad Chalasani, Art Charlesworth, Julie Clark, Olivier Danvy, Rich Goodwin, Gary Greenfield, Jonathan Hardwick, Maurice Herlihy, Susan Hinrichs, Puneet Kumar, Mark Leone, Mark Lillibridge, Kevin Lynch, David MacQueen, Yasuhiko Minamide, Scott Nettles, Brian Noble, Chris Okasaki, Sue Older, Scott Reilly, Norman Sobel, David Steere, Andrew Tolmach, and Mark Wheeler.

Of course, my mother and father deserve the most thanks and credit (especially for the ZX81!). Thanks everyone!

Greg Morrisett
December, 1995

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of my thesis is to show that types can and should be used throughout implementations of modern programming languages. More specifically, I claim that, through the use of *type-directed translation* and *dynamic type dispatch* (explained below), we can compile polymorphic, garbage-collected languages, such as Standard ML [90], without sacrificing natural data representations, efficient calling conventions, or separate compilation. Furthermore, I claim that a principled language implementation based on types lends itself to proofs of correctness, as well as tools that automatically verify the integrity of the implementation. In short, compiling with types yields both safety and performance.

Traditionally, compilers for low-level, monomorphic languages, such as C and Pascal, have taken advantage of the invariants guaranteed by types to determine data representations, alignment, calling conventions, register selection and so on. For example, when allocating space for a record, a C compiler can determine the size of the record from its type. When allocating a register for a variable of type double, a C compiler will use a floating point register instead of a general purpose register. Some implementations take advantage of types to support tag-free garbage collection [23, 119, 6] and so-called "conservative" garbage collection [21]. Types are also used to support debugging, printing and parsing, marshaling, and other means of traversing a data structure.

In addition to directing implementation, types are useful for proving formal properties of programs. For instance, it is possible to prove that every term in the simply-typed λ-calculus terminates. Similarly, it is possible to show that there is no closed value in the Girard-Reynolds polymorphic λ-calculus with the type $\forall\alpha.\alpha$. Compilers can take advantage of these properties to produce better code. For instance, a compiler can determine that a function, which takes an argument of type $\forall\alpha.\alpha$, will never be called simply because there are no values of the argument type. Therefore, the compiler can safely eliminate the function.

Types are also useful for proving relations between programs. In particular, types are useful for showing that two programs are equivalent, in the sense that they compute equivalent values. This provides a powerful mechanism for proving that a compiler is correct: Establish a set of simulation relations based on types, and then show that every source program and its translation are in the relation given by the source program's type.

Unfortunately, two obstacles have prevented language implementors from taking full advantage of types. The first obstacle is that implementors have lacked a sufficiently powerful, yet convenient framework for formally expressing their uses of types. Instead, implementors rely upon informal, *ad hoc* specifications of typing properties. This in turn prevents the implementor from proving formal properties based on types, and from recognizing opportunities for better implementation techniques within a compiler or runtime system.

A good example of this issue comes from the literature on type-based, tag-free garbage collection, where we find many descriptions of clever schemes for maintaining type information at run time to support memory management. Many of the approaches are surprisingly difficult to implement and rely upon very subtle typing properties. Yet, few if any of these descriptions are formal in any sort of mathematical sense. Indeed, the basic definitions of program evaluation and what it means for a value to be garbage are at best described informally, and at worst left unstated. Consequently, we have no guarantee that the algorithms are in any way correct. Practically speaking, this keeps us from modifying or adapting the algorithms with any assurance, simply because the necessary invariants are left implicit.

The second obstacle keeping implementors from fully taking advantage of types is that types have become complex, relative to the simple monomorphic type systems of C and Pascal. To support better static type checking, abstraction, code reuse, separate compilation, and other software engineering practices, we have evolved from using simple monomorphic types, to using the complex, modern types of Standard ML (SML), Haskell, and Quest. These modern types include polymorphic types, abstract types, object types, module types, qualified types, and even dependent types. These kinds of types have one thing in common: They include component types that are *unknown* at compile time. In fact, many of these types contain components that are *variable* at run time.

Most of the type-based implementation techniques used in compilers for C and Pascal rely critically upon knowing the type of every object at compile time. Implementors have lacked a sufficiently general approach for extending these techniques to cover unknown or variable types. Because of this, compilers for languages like SML, which have variable types, have traditionally ignored type information, and treated the language as if it was uni-typed. Consequently and ironically, implementations of modern languages suffer performance problems because they provide advanced types, but fail to take advantage of simple types.

The purpose of this thesis is to remove these two obstacles and open the path for language implementors to take full advantage of types. To address the first obstacle, lack of formalism, I demonstrate how to formalize key compilation phases and a run-time system using a methodology called *type-directed translation*. This methodology provides a unifying framework for specifying and proving the correctness of a compiler that takes full advantage of types. However, to compile languages like SML, the methodology of type-directed translation requires some formal mechanism for dealing with the second obstacle — variable types.

To address variable types, I provide a new compilation technique, *dynamic type dispatch*, that extends traditional approaches for compiling monomorphic languages to handle modern types. In principle, dynamic type dispatch has none of the drawbacks of previous approaches, but it introduces some issues of its own. In particular, to take full advantage of dynamic type dispatch, we must propagate type information through each phase of a compiler. Fortunately, type-directed translation provides a road map for achieving this goal.

In short, the two contributions of this thesis, type-directed translation and dynamic type dispatch, are equally important because they rely critically upon each other. To demonstrate the practicality of these two techniques, I (with others) have constructed a compiler for SML called TIL. TIL, which stands for *Typed Intermediate Languages*, takes advantage of both type-directed translation and dynamic type dispatch to provide natural representations and calling conventions. The type-directed transformations performed by TIL reduce running times by roughly 40% and heap allocation by 50%.

In addition to type-directed translation and dynamic type dispatch, TIL employs a set of conventional functional language optimizations. These optimizations account for much of the good performance of TIL, in spite of the fact that they operate on statically-typed intermediate languages. Indeed, TIL produces code that is roughly twice as fast as code produced by the SML/NJ compiler [12], which is one of the best existing compilers for Standard ML.

The rest of this chapter serves as an overview of the thesis. In Section 1.1, I give an overview of type-directed translation. In Section 1.2, I discuss the problem of compiling in the presence of variable types, discuss previous approaches to this problem, and show why these solutions are inadequate. In Section 1.3, I give an overview of dynamic type dispatch and discuss why it is a superior to previous approaches of compiling in the presence of variable types. In Section 1.4, I discuss the key issue of using dynamic type dispatch in conjunction with type-directed translation — how to type check a language that provides dynamic type dispatch. Finally, in Section 1.5, I give a comprehensive overview of the rest of the thesis.

1.1 Type Directed Translation

A compiler transforms a program in a source language into a program in a target language. Usually, we think of the target language as more "primitive" than the source language according to functionality. As an example, consider the compilation of SML programs to a lower-level language such as C. We consider C "lower-level" because SML provides features that C does not directly provide, such as closures, exceptions, automatic memory management, algebraic data types, and so forth. These high-level constructs must be encoded into constructs that C does support. For instance, closures can be encoded in C as a `struct` containing both a pointer to a C function and a pointer to another `struct` that contains values of the free variables in the closure.

It is a daunting task to compile a high-level language, such as SML, to a relatively low-level language, such as C, and even more daunting to compile to an extremely low-level language, such as machine code. The only feasible approach to overcome the complexity is to break the task into a series of simpler compilers that successively map their source language to closely related, but slightly simpler target languages. Taking the sequential composition of these simpler compilers yields a compiler from the original source language to the final target language. Decomposing a compiler into a series of simpler compilers has an added benefit: Correctness of the entire compiler can be established by proving the correctness of each of the simpler compilers.

The initial task a compiler writer faces is deciding *how* to break her compiler into smaller, more manageable compilation steps. She must decide what language feature(s) to eliminate in each step of compilation and she must develop a strategy for how this is to be accomplished. Next, for each stage of compilation she must design and specify the intermediate target languages. This includes formally specifying a dynamic semantics so that we know precisely what each target language construct means. Then, the compiler writer must formulate a precise, but sufficiently high-level description of an algorithm that maps the source language to the target language for each stage. Finally, the compiler writer must prove each of the translation algorithms correct. A translation is correct if when we run the source program (using the source language dynamic semantics) and we run the translation of the program (using the target language dynamic semantics), then we get "equivalent" answers. For simple answers, such as strings or integers, "equivalent" usually means syntactic equality. However, weaker, semantic notions of equivalence are needed to relate more complex objects such as functions.

In this thesis, I demonstrate this methodology by deriving key parts of a compiler from a simple ML-like functional language to a relatively low-level language that makes representations, calling conventions, closures, allocation, and garbage collection explicit. I formulate the translation as a series of *type-directed* and *type-preserving* maps. By type-directed, I mean that the source language of each stage is statically typed and source types are used to guide the compilation at every step. For instance, given a generic structural equality operation at the source level, we can select the code that implements the operation according to the type assigned to the arguments of the operation. If that type is int, we use integer equality, and if the type is float, we use floating point equality, and so on.

By a type-preserving translation, I mean the following: first, both the source and the target languages are typed. Second, a type-preserving translation specifies a type translation in addition to a term translation. Third, if a source expression e has type τ, the term translation of e is e', and the type translation of τ is τ', then e' has type τ'. Therefore, assuming the input is well-typed, so is the output.

By using a typed target language in addition to a typed source language, we ensure that any later translations in the compiler can continue to take advantage of types for their own purpose. For example, in Chapter 7, I take advantage of these types to implement garbage collection.

In type-directed translation, we not only use types to guide the translation, but we also use types to argue that the translation is correct. Indeed, it is the presence of types that allows us to define what it means for the translation to be correct! Thus, the contribution of types to the compilation process is many fold: We use types to select appropriate representations, calling conventions, and primitive operations, and we use types to prove that the compiler is correct.

1.2 The Issue of Variable Types

Modern programming languages, such as C++, CLU, Modula-3, Ada, Standard ML, Eiffel, and Haskell all provide type systems that are much more expressive than the simple, monomorphic systems of C and Pascal. In particular, each of these languages supports at least one notion of *unknown* or *variable* type. Variable types arise in conjunction with two key language advances: abstract data types and polymorphism. These features are the building blocks of relatively new language features including modules, generics, objects, and classes.

The SML code in Figure 1.1 provides an example use of an unknown type. The code implements a merge sort on a list of values of uniform, but unknown type, denoted by α. The function `sort` takes a predicate `lt` (less-than) of type α `*` α `-> bool`, a list of α values, and produces a list of α values. We can apply the sort function to a list of integers, passing integer less-than as the comparison operator:

```
sort (op < : int*int->bool) [5,7,2,1,3,9,10]
```

Alternatively, we can apply the sort function to a list of floating point values, passing floating point less-than as the comparison operator:

```
sort (op < : real*real->bool) [138.0,3.1415,4.79]
```

```
fun sort (lt:α*α->bool) (l:α list) : α list =
        let fun merge (nil,l) = l
              | merge (l,nil) = l
              | merge (x::tx,y::ty) =
                if lt(x,y) then x :: merge(tx,y::ty)
                else y :: merge(x::tx,ty)
            fun split ([],one,two) = (sort lt one,sort lt two)
              | split (x::tl,one,two) = split (tl,two,x::one)
        in
            case l of
              _::_::_ => merge (split(l,[],[]))
            | _ => l
        end
```

Figure 1.1: A Polymorphic Merge Sort Function

In fact, we can pass a list of any type to the sort routine provided we have an appropriate comparison operation for that type. By abstracting the component type of the list as a type variable, we can write the sort code once and use it at as many types as we like. For monomorphic languages like Pascal, which strictly enforce types, a sort implementation must be copied for each instantiation of α. This can waste code space and make program maintenance more difficult. For instance, if we find a bug in the sort implementation, then we must make sure to eliminate the bug in all copies.

The ability to use code at different types, as in the preceding sort example, is usually called *polymorphism*, meaning “many changing”. Polymorphism is closely related to the notion of abstract data types (ADTs). ADTs are objects that implement some data type and its corresponding operations, but hold the representation of the data type and the implementation of the operations abstract. For example, the following SML code implements a stack of integers as an ADT via the `abstype` mechanism:

```
abstype stack = Stack of int list
with
   val empty = Stack nil
   fun push (x, Stack s) = Stack (x :: s)
   exception Empty
   fun pop (Stack nil) = raise Empty
     | pop (Stack (x :: s)) = (x, Stack s)
end
```

The type system of SML prevents the client of an `abstype` from examining the representation of the abstracted type. Hence, if we try to use a stack as if it is an integer list, we will get a type error. A key advantage of abstracting the type is that, in principal, we can separately compile the definition of the ADT from its uses. Furthermore, we should be able to change the representation of the abstracted type without having to recompile clients of the abstraction. For example, we could use a vector to represent the stack of integers instead of a list.

But how do we compile in the presence of variable types? Consider, for example, the following issues:

1. Functions can have arguments of unknown type:

   `fun id (x:`α`) : `α` = x`

   Since α can be instantiated to any type, what register should we use to pass the argument to `id`? Should we use a floating point or general purpose register?

2. The following function creates a record of values whose types are unknown:

   `fun foo (x:`α`,y:`β`) = (x,y,x,y)`

   How much space should we allocate for the data structure? How do we align the components of the record to support efficient access? Should we add padding? Should we pack the fields of the record?

3. In languages such as SML, n-argument functions are represented by functions taking a single n-tuple as an argument. This makes the language uniform and simplifies the semantics. But for efficiency, we want to "flatten" a tuple argument into multiple arguments. If the argument to a function has a variable type, then how do we know if we should flatten the argument?

4. When compiling an ad-hoc polymorphic operation such as structural equality (e.g., `eq`(e_1, e_2)) and the type of the arguments is a variable, what code should we generate? Should we generate an integer comparison, floating point comparison, or code that extracts components and recursively compares them for equality?

5. How do we determine the size and pointers of a value of unknown type so that we can perform tracing garbage collection?

In this section, I explore existing solutions to these issues and discuss their relative strengths and weaknesses. As I will show, none of the existing approaches preserves all of the following desirable implementation properties:

- separate compilation

- natural representations and calling conventions
- fully polymorphic definitions or fully abstract data types

1.2.1 Previous Approach: Eliminate Variable Types

The easiest way to implement a language with variable types is to contrive either the language or the implementation so that all of the variable types are eliminated *before* compilation begins. This allows us to use a standard, monomorphic compiler.

The "elimination" approach has been used in various guises by implementations of C++ [114], Ada [121], NESL [20], and Gofer [74] to support ADTs and polymorphism.

- When defining a new class in C++, the definition is placed in a "`.h`" file. The definition is `#included` by any client code that wishes to use the abstraction. Hence, the compiler can always determine the representation of an abstract data type. The type system enforces the abstracted type within the client, but the first stage of compilation eliminates the abstracted type variable, and replaces it with its implementation definition.
- When defining an ADT via the package mechanism of Ada, it is sometimes necessary to expose the representation of the ADT in the interface of the package. This representation is "hidden" in a private part of the interface. Again, the type system of the language enforces the abstraction, but the first stage of compilation replaces the abstract type variable with the implementation representation.
- NESL is a programming language for parallel computations that allows programmers to define polymorphic functions [20]. However, the NESL implementation delays compiling any polymorphic definitions. Instead, whenever a polymorphic function is instantiated with a particular type, the type is substituted for the occurrences of the type variable within the polymorphic code. The resulting monomorphic code is compiled. A caching scheme is used to minimize code duplication.
- Gofer, a dialect of Haskell, provides both polymorphism and type classes. Mark Jones constructed an implementation that, like the NESL implementation, performs all polymorphic instantiation at compile time [73].

Unfortunately, the "eliminate variable types" approach has many drawbacks. One drawback is that polymorphic code is never shared. Instead, each polymorphic definition is copied at least once for each unique instantiation. This can have a serious effect on both compile times and instruction cache locality. For C++, the definitions in the "`.h`" file must be processed each time a client is compiled. Newer compilers attempt to cache the results of this processing in a separate file precisely to avoid this compilation overhead.

The caching scheme used by NESL ensures that exactly one copy is made for a given type, but no code is actually shared across different instantiations. Both caching schemes introduce a coherence problem: When the polymorphic definition is updated, the cached definitions must be discarded. For languages like Gofer, Haskell, and SML, which provide nested polymorphic definitions, it is possible that the number of copies of a polymorphic definition could grow exponentially with the number of type variables in the definition. (However, Jones reports that this does not occur in practice [73].)

Even if code size, compile times, and instruction cache locality were not an issue, the "eliminate" approach sacrifices separate compilation of an ADT or polymorphic definition from its uses. For example, if we change the implementation of an ADT implemented using an Ada package, then we must also change the private portion of the package specification. Since all clients depend upon this specification, changing the ADT implementation requires that all clients be recompiled. Similarly, a simple change to a class definition in C++ can require the entire program to be recompiled.

Increasingly, we are moving away from a world where we have access to all of the source files of a program, and where we can "batch" process the compilation, linking, and loading of a program. For example, vendor-supplied, dynamically-linked libraries (e.g., Xlib, Tk) are now the norm instead of the exception. Often, it is impossible to get the source code for such libraries and it is prohibitively time-consuming to recompile them for each application, especially during development. We now have languages such as Java [51] and Obliq [27, 28] where objects and code are dynamically transmitted from one machine to another via a network, compiled to a native representation, and then dynamically linked into a running program. Hence, the ability to compile program components separately is becoming increasingly important and any compilation methodology must provide some level of support for separate compilation.

Finally, for many newer programming languages, it is simply impossible to eliminate all polymorphism or all ADTs at compile time. Consider, for example, a language that supports first-class polymorphic definitions. Such objects can be placed in data structures, passed as arguments to functions, and so forth. Thus, determining all of the types that instantiate a given definition becomes in general undecidable.

1.2.2 Previous Approach: Restrict Representations

A different approach to compiling in the presence of variable types is to restrict the types that can instantiate a type variable. This approach, known as *boxing*, restricts type variables so that, no matter what the actual type is, the representation of the value has the same size. As an example, Modula-3 allows only *pointer* types (e.g., $\mathsf{ptr}[\tau]$) to be used as the implementation of an abstract type. Assuming all pointers are the same size[1], we can always allocate a data structure containing values of type ptr[t], even if t is unknown. Similarly, we know that such values will be passed in general purpose as opposed to floating point registers.

Languages like SML do not make a restriction on polymorphism or variable types at the source level, but almost all implementations use such a restriction in compilation. In essence, they perform a type-directed translation that maps variable types (t) to pointer types (ptr[t]). Unfortunately, a naive translation that always maps type variables to pointer types is not type-preserving. Consider for example the polymorphic identity function and its use at some type:

```
let fun id (x:α) : α = x
in
    (id : float->float) 3.1415
end
```

The naive translation yields:

```
let fun id (x:ptr[α]) : ptr[α] = x
in
   (id : ptr[float]->ptr[float]) 3.1415
end
```

However, this translation is ill-typed because `id` takes a pointer as an argument, but the literal `3.1415` is a floating point value. Since the translation is ill-typed, the rest of the compiler will produce erroneous code. For example, the translation of the application of `id` to the floating point value may place the argument into a floating point register, whereas the code of the function will be translated with the expectation that the argument is in a general purpose register.

The problem is that in general, it is impossible to tell whether or not a value will be passed to a polymorphic function. If a value is passed as an argument of unknown type to some routine, then the value must be boxed (i.e., represented as a pointer.) Because it is impossible to tell whether or not a value will be passed to a polymorphic function, most ML compilers, including Poly/ML [88], Bigloo [107], Caml [126], and older versions of SML/NJ [9], box *all* objects.

Boxing supports separate compilation and dynamic linking, but unfortunately, it consumes space and time because of the extra indirection that is introduced. For example, to support polymorphic array operations, an array of floating point values must be represented by an array of pointers to singleton records that contain the actual floating point values (see Figures 1.2 and 1.3). An extra word is used as the pointer for each array

[1]Even this is a dangerous assumption in many environments, including MS-DOS versions of Borland C, where distinctions are made between "near" and "far" pointers.

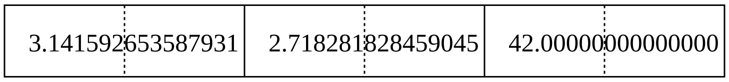


Figure 1.2: Natural Representation of a Floating Point Array

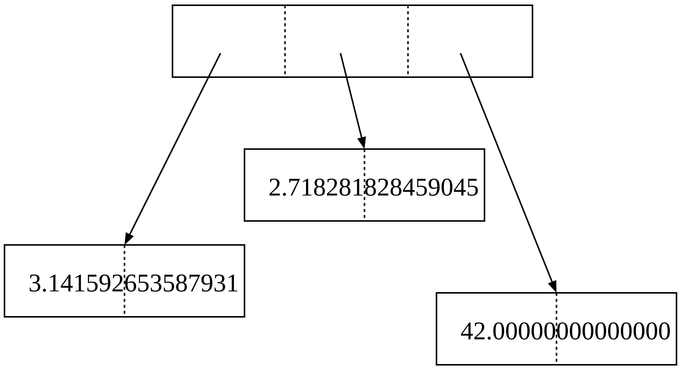


Figure 1.3: Boxed Representation of a Floating Point Array

element. For systems such as SML/NJ that use tagging garbage collection, an additional tag word is required for each element. Hence, the polymorphic array can consume twice as much space as its monomorphic counterpart. Furthermore, accessing an array element requires an additional memory operation. As memory speeds become slower relative to processor speeds, this extra memory operation per access becomes more costly. Finally, in the presence of a copying garbage collector, the elements of the array could be scattered across memory. This could destroy the spatial locality of the array, resulting in increased data cache misses and even longer access times.

In addition to causing performance problems, boxing also interferes with interoperability. As with tags to support garbage collection, adding extra indirection can impede communication with systems that do not use boxing. In particular, it becomes difficult to communicate with libraries, runtime services, and operating system services, because they tend to be written in low-level languages such as C or Fortran that do not provide variable types. Extra code must be written to “marshal” a data structure from its boxed representation to the representation used by the library, runtime, or operating system.

Finally, although boxing addresses many of the issues of compiling in the presence of variable types, it does not help us eliminate overloaded operators (such as structural equality) or perform garbage collection. Hence, standard implementations of SML both *box* and *tag* all values to support their advanced language features. As a direct result, the quality of the code emitted by most SML compilers, even when these features are not used, is far below the quality of compilers for languages like C or Fortran.

1.2.3 Previous Approach: Coercions

Because boxing and tagging are so expensive, a great deal of research has gone into minimizing these costs [75, 81, 64, 65, 102, 110]. A particularly clever approach was suggested by Xavier Leroy for call-by-value languages based on the ML type system [81]. The fundamental idea is to compile monomorphic code in exactly the same way that it is compiled in the absence of variable types, and to compile polymorphic code assuming that variables of unknown type are boxed and tagged. As we showed in Section 1.2.2, this results in a type mismatch when a polymorphic object is instantiated. Leroy's solution is to apply a *coercion* to the polymorphic object to mitigate this mismatch. The coercion is based on the type of the object and the type at which it is being instantiated. A fascinating property of Leroy's solution is that, for languages based on the ML type system, the appropriate coercion to apply in a given situation can always be determined at compile time.

As an example, the naive boxing translation of the identity function produced the following incorrect code:

```
let fun id (x:ptr[α]) : ptr[α] = x
in
   (id : ptr[float]->ptr[float]) 3.1415
end
```

Leroy's translation fixes the mismatch by applying a *boxing* coercion to the argument of the polymorphic function and an *unboxing* coercion to the result:

```
let fun id (x:ptr[α]) : ptr[α] = x
in
   unbox[float]
    (id : ptr[float]->ptr[float]) (box[float](3.1415))
end
```

Assuming `box` and `unbox` convert a value to and from a pointer representation and add any necessary tags, the resulting code is operationally correct. In general, a polymorphic object of type $\forall\alpha.\tau[\alpha]$ is compiled with the type $\forall\alpha.\tau[\mathsf{ptr}[\alpha]]$. When instantiated with some type τ', the object has the type $\tau[\mathsf{ptr}[\tau']]$, but the object is expected to have the type $\tau[\tau']$. A coercion is applied to the instantiated object to correct the mismatch. The coercion is calculated via a function S that maps the type scheme $\tau[\alpha]$ and the instantiating type τ' to a term as follows:

$$\begin{array}{rcl}
S[\alpha,\tau'] & = & \lambda x.\,\mathtt{unbox}[\tau'](x) \\
S[\mathsf{int},\tau'] & = & \lambda x.\,x \\
S[\mathsf{float},\tau'] & = & \lambda x.\,x \\
S[\langle\tau_1\times\cdots\times\tau_n\rangle,\tau'] & = & \lambda x.\,\mathtt{let}\ x_1{=}\pi_1\,x,\cdots,x_n{=}\pi_n\,x \\
 & & \quad\mathtt{in}\ \langle S[\tau_1,\tau'](x_1),\cdots,S[\tau_n,\tau'](x_n)\rangle \\
S[\tau_1\to\tau_2,\tau'] & = & \lambda f.\,\lambda x.\,S[\tau_2,\tau'](f\ (G[\tau_1,\tau']x))
\end{array}$$

The definition of S at arrow types uses the dual coercion function G:

$$\begin{array}{rcl}
G[\alpha,\tau'] & = & \lambda x.\,\mathtt{box}[\tau'](x) \\
G[\mathsf{int},\tau'] & = & \lambda x.\,x \\
G[\mathsf{float},\tau'] & = & \lambda x.\,x \\
G[\langle\tau_1\times\cdots\times\tau_n\rangle,\tau'] & = & \lambda x.\,\mathtt{let}\ x_1{=}\pi_1\,x,\cdots,x_n{=}\pi_n\,x \\
 & & \quad\mathtt{in}\ \langle G[\tau_1,\tau'](x_1),\cdots,G[\tau_n,\tau'](x_n)\rangle \\
G[\tau_1\to\tau_2,\tau'] & = & \lambda f.\,\lambda x.\,G[\tau_2,\tau'](f\ (S[\tau_1,\tau']x))
\end{array}$$

The coercions generated by S and G deconstruct a value into its components until we reach a base type or a type variable. The coercion at a base type is the identity but the coercion at a variable type requires either boxing or unboxing that component. Once the components have been coerced, the aggregate value is reassembled. Hence, it is fairly easy to show that:

$$\begin{array}{rcl}
S[\tau[\alpha],\tau'] & : & \tau[\mathsf{ptr}[\tau']]\to\tau[\tau'] \\
G[\tau[\alpha],\tau'] & : & \tau[\tau']\to\tau[\mathsf{ptr}[\tau']]
\end{array}$$

and thus S and G appropriately mitigate the type mismatch that occurs at polymorphic instantiation.

The coercion approach offers the best mix of features from the set of solutions presented thus far. In particular, as with full boxing, it supports separate compilation and code sharing. Unlike full boxing, monomorphic code does not have to pay the penalties of boxing and tagging. Leroy found that his coercion approach cut execution time by up to a factor of two for some benchmarks run through his Gallium compiler, notably numeric codes that manipulate many integer or floating point values. However, for at least one contrived program with a great deal of polymorphism, the coercion approach slowed the program by more than a factor of two [81]. Nevertheless, his coercion approach has an attractive property: *You pay only for the polymorphism you use.*

Other researchers have also found that eliminating boxing and tagging through coercions can cut execution times and allocation considerably. For instance, Shao and Appel were able to improve execution time by about 19% and decrease heap allocation by 36% via Leroy-style coercions for their SML/NJ compiler [110]. However, much of their improvement (11% execution time, 30% of allocation) comes by performing a type-directed flattening of function arguments as part of the coercion process.

Unfortunately, the coercion approach has some practical drawbacks: First, the coercions operate by deconstructing a value, boxing or unboxing some components, and then building a copy of the value out of the coerced components. Building a copy of a large data structure, such as a list, array, or vector, requires mapping a coercion across the whole data structure. Such coercions can be prohibitively expensive and applying them may well outweigh the benefits of leaving the data structure unboxed. Second, in the presence of recursive types (ML data types), refs, or arrays, not only must the components corresponding to type variables be boxed, but *their* components must be recursively boxed [81]. Third, and perhaps most troublesome, it is difficult if not impossible to make a copy of a mutable data structure such as an array. The problem is that updates to the copy must be reflected in the original data structure and vice versa. Hence, it is impossible to apply a coercion to refs or arrays and consequently, the components of such data structures must always be boxed.

It is possible to represent refs and arrays as a pair of "get" and "set" functions whose shared closure contains the actual ref cell or array. Then the standard functional coercions can be applied to the get and set operations to yield a coerced mutable data structure. However, having to perform a function call to access a component of an array can easily offset any benefits from leaving the array unboxed.

Finally, the coercion approach to variable types is simply a stop-gap measure. It takes advantage of certain properties of the ML type system – notably the lack of first-class polymorphic objects – to ensure that the appropriate coercion can always be calculated at compile time. This approach breaks down when we move to a language with first-class polymorphism.

1.3 Dynamic Type Dispatch

There is an approach for compiling in the presence of variable types, first suggested by the Napier '88 implementation [97], which avoids the drawbacks of boxing or coercions without sacrificing separate compilation. The idea is to delay deciding what code to select until types are known. This is accomplished by passing types that are unknown at compile-time to primitive operations. Then, the operations can analyze the type in order to select and dispatch to the appropriate code needed to manipulate the natural representation of an object. I call such an approach *dynamic type dispatch*.

For example, a polymorphic subscript function on arrays might be compiled into the following pseudo-code:

```
sub = Λα. typecase α of
            int => intsub
          | float => floatsub
          | ptr[σ] => ptrsub[σ]
```

assuming the following operations, where we elide the "$\mathsf{ptr}[-]$" around arrow and array types for clarity:

$$
\begin{array}{rcl}
\texttt{intsub} & : & [\mathsf{intarray}, \mathsf{int}] \to \mathsf{int} \\
\texttt{floatsub} & : & [\mathsf{floatarray}, \mathsf{int}] \to \mathsf{float} \\
\texttt{ptrsub}[\mathsf{ptr}[\sigma]] & : & [\mathsf{ptrarray}[\mathsf{ptr}[\sigma]], \mathsf{int}] \to \mathsf{ptr}[\sigma]
\end{array}
$$

Here, `sub` is a function that takes a *type* argument (α), and then performs a case analysis to determine the appropriate specialized subscript function that should be returned. For example, `sub`[int] returns the integer subscript function that expects an array of integers, whereas `sub`[float] returns the floating point subscript function that expects a double-word aligned array of floating point values. All other types are pointers, so we assume the array has boxed components and thus `sub` returns the boxed subscript function at the appropriate large type.

If the `sub` operation is instantiated with a type that is known at compile-time (or link-time), then the overhead of the case analysis can be eliminated by duplicating and specializing the definition of `sub` at the appropriate type. For example, the source expression

```
sub(x,4) + 3.14,
```

will be compiled to the target expression

```
sub[float](x,4) + 3.14,
```

since the result of the `sub` operation is constrained to be a float. If the definition of `sub` is inlined into the target expression and some simple reductions are performed, this yields the optimized expression:

```
floatsub(x,4) + 3.14.
```

Like the coercion approach to compiling with variable types, dynamic type dispatch supports separate compilation and allows us to pay only for the polymorphism that we use. In particular, monomorphic code can be compiled as if there are no variable types. Furthermore, unlike coercions, dynamic type dispatch supports natural representations for large data structures (such as lists, arrays, or vectors) and for mutable data structures (such as arrays). Instead of coercing the values of the data structures, we coerce the behavior of the operations. Hence, we do not have to worry about keeping copies of a mutable data structure coherent. As a result, dynamic type dispatch provides better *interoperability* than any of the previously tried solutions, without sacrificing separate compilation.

As I will show, dynamic type dispatch also supports tag-free overloaded operations. For example, we can code an ML-style polymorphic equality routine by dispatching on a type:

```
typerec eq[int] = λ(x,y).=_int(x,y)
      | eq[float] = λ(x,y).=_float(x,y)
      | eq[ptr[⟨τ1 × τ2⟩]] =
          λ(x,y). eq[τ1](π1 x,π1 y) andalso eq[τ2](π2 x,π2 y)
      | eq[ptr[τ1 → τ2]] = λ(x,y). false
```

The same approach can be used to dynamically flatten arguments into registers, dynamically flatten structs and align their components, and so on. Finally, by passing unknown types to the garbage collector at run-time, dynamic type dispatch supports tag-free garbage collection.

In short, dynamic type dispatch provides a smooth transition from compilers for monomorphic languages to compilers for modern languages with variable types.

1.4 Typing Dynamic Type Dispatch

If we are to use types to support register allocation, calling conventions, data structure layout, and garbage collection, we must propagate types *through* compilation to the stages where these decisions are made. Many of these decisions are made after optimization or code transformations, so it is important that we can propagate type information to as low a level as possible.

An intermediate language that supports run-time type dispatch allows us to express source primitives, such as array subscript or polymorphic equality, as terms in the language. This exposes the low-level operations of the source primitive to optimization and transformations that may not be expressible at the source level.

If we are to use an intermediate language that supports run-time type dispatch, we must be able to assign a type to terms that use `typecase`. But what type should we give a term such as `sub`, shown previously? We cannot use a *parametric* type such as $\forall\alpha.[\mathsf{array}[\alpha],\mathsf{int}] \to \alpha$, because instantiating `sub` with `int` for instance, yields the `intsub` operation of type $[\mathsf{intarray},\mathsf{int}] \to \mathsf{int}$ which is not an instantiation of the parametric type.

My approach to this problem is to consider a type system that provides type dispatch at the *type* level via a "Typecase" construct. For example, the `sub` definition can be assigned a type of the form:

$$\forall\alpha.[\mathsf{SpclArray}[\alpha],\mathsf{int}] \to \alpha$$

where the specialized array constructor SpclArray is defined using Typecase as follows:

```
SpclArray[α] = Typecase α of
                  int => intarray
                | float => floatarray
                | ptr[σ] => ptrarray[ptr[σ]]
```

The definition of the constructor parallels the definition of the term: If the parameter α is instantiated to int, the resulting type is $\mathsf{intarray}$; if the parameter is instantiated to float, the resulting type is $\mathsf{floatarray}$.

In this thesis, I present a formal calculus called λ_i^{ML} that provides run-time type passing and type dispatch operations. The calculus is intended to provide the formal underpinnings of a target language for compiling in the presence of variable types. I prove two important properties regarding λ_i^{ML}: The type system is *sound* and type checking is *decidable*.

In its full generality, λ_i^{ML} allows types to be analyzed not just by case analysis (i.e., `typecase`), but also via primitive recursion. This allows more sophisticated transformations to be coded *within* the target language, yet type checking for the target language remains decidable.

An example of a more sophisticated translation made possible by primitive recursion is one where arrays of pointers to pairs are represented as a pointer to a pair of arrays. For example, an array of $\mathsf{ptr}[\langle\mathsf{int}\times\mathsf{float}\rangle]$ is represented as a pointer to a pair of an $\mathsf{intarray}$ and a $\mathsf{floatarray}$. This representation allows the integer components of the array to be packed and allows the floating point components to be naturally aligned. It also saves $n-1$ words of indirection for an array of size n, since pairs are normally boxed. The subscript operation for this representation is defined using a recursive $\mathsf{typecase}$ construct called $\mathsf{typerec}$ in the following manner:

$$
\begin{array}{ll}
\texttt{typerec} & \texttt{sub[}\mathsf{int}\texttt{] = intsub} \\
& \texttt{| sub[}\mathsf{float}\texttt{] = floatsub} \\
& \texttt{| sub[}\mathsf{ptr}[\langle\tau_1\times\tau_2\rangle]\texttt{] = }\lambda\texttt{[}\langle\texttt{x,y}\rangle\texttt{,i].}\langle\texttt{sub[}\tau_1\texttt{]x,sub[}\tau_2\texttt{]y}\rangle \\
& \texttt{| sub[}\mathsf{ptr}[\sigma]\texttt{] = ptrsub[}\mathsf{ptr}[\sigma]\texttt{]}
\end{array}
$$

If sub is given a product type, $\mathsf{ptr}[\langle\tau_1\times\tau_2\rangle]$, it returns a function that takes a pair of arrays ($\langle\texttt{x,y}\rangle$) and an index (`i`), and returns the pair of values from both arrays at that index, recursively calling the sub operation at the types τ_1 and τ_2.

The type of this sub operation is:

$$\forall\alpha.[\mathsf{RecArray}[\alpha],\mathsf{int}]\rightarrow\alpha$$

where the recursive, specialized array constructor $\mathsf{RecArray}$ is defined using a type-level "Typerec":

$$
\begin{array}{lrcl}
\mathsf{Typerec} & \mathsf{RecArray}\,[\mathsf{int}] & = & \mathsf{intarray} \\
& \mid \mathsf{RecArray}\,[\mathsf{float}] & = & \mathsf{floatarray} \\
& \mid \mathsf{RecArray}\,[\mathsf{ptr}[\langle\tau_1\times\tau_2\rangle]] & = & \mathsf{ptr}[\langle\mathsf{RecArray}[\tau_1]\times\mathsf{RecArray}[\tau_2]\rangle] \\
& \mid \mathsf{RecArray}[\mathsf{ptr}[\sigma]] & = & \mathsf{ptrarray}[\mathsf{ptr}[\sigma]]
\end{array}
$$

Again, the definition of the constructor parallels the definition of the sub operation. If the parameter is instantiated with int, then the resulting type is $\mathsf{ptr}[\mathsf{intarray}]$. If the parameter is instantiated with $\mathsf{ptr}[\langle \tau_1 \times \tau_2 \rangle]$, then the resulting type is the product of $\mathsf{RecArray}[\tau_1]$ and $\mathsf{RecArray}[\tau_2]$.

1.5 Overview of the Thesis

In this thesis, I show that we can take advantage of types to compile languages with variable types, without losing control over data representations or performance. In particular, I show that dynamic type dispatch can be used to support efficient calling conventions and native representations of data without sacrificing efficient monomorphic code, separate compilation, or tag-free garbage collection. I also show how key pieces of a standard functional language implementation must be extended to accommodate dynamic type dispatch. For instance, I show that representation analysis, closure conversion, and garbage collection can all be extended to work with and take advantage of dynamic type dispatch.

A significant contribution of my thesis is that I formulate these aspects of language implementation at a fairly abstract level. This allows me to present concise proofs of correctness. Even if we ignore dynamic type dispatch, exhibiting compact formulations and correctness arguments for representation analysis, closure conversion, and garbage collection is a significant contribution.

The TIL compiler demonstrates not only that dynamic type dispatch is a viable technique for compiling in the presence of variable types, but also that type-directed compilation does not interfere with standard optimization, such as inlining (β-reduction), common sub-expression elimination, and loop invariant removal. Also, TIL demonstrates that these standard optimizations are, for the most part, sufficient to eliminate the overheads of dynamic type dispatch for an SML-like language.

I now briefly outline the remainder of the thesis: In Chapter 2, I present a simple, core polymorphic source language called Mini-ML. I define the syntax, dynamic semantics, and static semantics of the language. I also state the key properties of the static semantics for the language. Readers familiar with ML-style polymorphism may want to skip this chapter, but compiler writers unfamiliar with formal semantic specifications may find this chapter illuminating.

In Chapter 3, I present a core intermediate language called λ_i^{ML}. This language provides the formal underpinnings of a calculus with dynamic type dispatch that is used in the subsequent chapters. I define the syntax, dynamic semantics, and static semantics of the language. In Chapter 4, I summarize the key semantic properties of the formal calculus, including decidability of type checking and soundness of the static semantics. Proofs of these properties follow for those interested in the underlying type theory. Compiler writers may want to skip these details.

In Chapter 5, I define a variant of λ_i^{ML}, called λ_i^{ML}-Rep, that makes calling conventions explicit. I show how to map Mini-ML to λ_i^{ML}-Rep. In the compilation, I show how to eliminate polymorphic equality and flatten function arguments, in order to demonstrate the power and utility of dynamic type dispatch. I establish a suitable set of logical simulation relations between Mini-ML and λ_i^{ML}-Rep and use them to prove the correctness of the translation. I then demonstrate how other language features can be implemented using dynamic type dispatch, including flattened data structures with aligned components, unboxed floating point arguments, Haskell-style type classes, and communication primitives.

In Chapter 6, I show how to map λ_i^{ML} to a language with closed code, explicit environments, and explicit closures. This translation, known as *closure conversion*, is important because it eliminates functions with free variables. The key difficulty is that I must account for free type variables as well as free term variables when producing the code of a closure and thus, environments must contain bindings for both value variables and type variables. Unlike most accounts of closure conversion, the target language of my translation is still typed. This allows me to propagate type information through closure conversion. In turn, this supports other type-directed transformations after closure conversion, as well as run-time type dispatch and tag-free garbage collection.

In Chapter 7, I provide an operational semantics for a monomorphic subset of closure converted λ_i^{ML} code. The semantics makes the heap and the stack explicit. I formalize garbage collection as any rewriting rule that drops portions of the heap without affecting evaluation. I specify and prove correct an abstract formulation of copying garbage collection based on the abstract syntax of terms (i.e., tags). I then show how types can be used to support tag-free garbage collection and prove that this approach is sound. Then I show how to extend the tag-free approach to a type-passing, polymorphic language like λ_i^{ML}.

In Chapter 8, I give an overview of TIL and the practical issues involved in compiling a real programming language to machine code in the presence of dynamic type dispatch. I also examine some aspects of the performance of TIL code: I compare the running times and space of TIL code against the code produced by SML/NJ. I also measure the impact that various type-directed translations have on both the running time and amount of data allocated.

Finally, in Chapter 9, I present a summary of the thesis, and discuss future directions.

Chapter 2

A Source Language: Mini-ML

In this chapter, I specify a starting source language, called Mini-ML, that is based on the core language of Standard ML [90, 31]. Although Mini-ML is a fairly limited language, it has many of the constructs that one might find in a conventional functional programming language, including integers and floating point values; first-class, lexically-scoped functions; tuples (records); and polymorphism. Indeed, I have designed Mini-ML so that it brings out the key issues one must address when compiling a modern language like SML.

The syntax of Mini-ML is given in Figure 2.1. There are four basic syntactic classes: monotypes, polytypes, values, and expressions. The monotypes of Mini-ML describe expressions and consist of type variables (t), base types including int, float and unit, and constructed types including $\langle \tau_1 \times \tau_2 \rangle$ and $\tau_1 \to \tau_2$. The monotypes are distinct from types that contain a quantifier ($\forall$). Polytypes, also referred to as type schemes, are either monotypes or prenex, universally-quantified monotypes. Type variables range over monotypes. Thus, polytypes are forbidden from instantiating a type variable.

Values consist of variables (x), integer and floating point literals (i and f), unit ($\langle\rangle$), pairs of values, term functions ($\lambda x{:}\tau.e$), and type functions ($\Lambda t_1, \cdots, t_n.e$). Term and type functions are sometimes referred to as term or type abstractions, respectively. Expressions contain variables, literals, unit, pairs of expressions, term functions, a structural equality operation ($\mathtt{eq}(e_1, e_2)$), a test for zero, projections ($\pi_i\, e$), and term applications ($e_1\, e_2$).

Expressions also include a `def` construct, which binds a variable to a value. Since values include type abstractions, this provides a means for binding a type abstraction to a variable, much like the `let` construct of SML. Finally, expressions include applications of values to monotypes ($v[\tau_1, \cdots, \tau_n]$). The typing rules, explained in Section 2.2 restrict v to be either a `def`-bound variable or a Λ-abstraction. Hence, the only thing that can be done with a Λ-abstraction is either bind it to a variable via `def` or apply it to some monotypes. This means that, as in SML, type abstractions are "second-class" because

$$
\begin{array}{llcl}
\text{(types)} & \tau & ::= & t \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{unit} \mid \langle \tau_1 \times \tau_2 \rangle \mid \tau_1 \to \tau_2 \\
\text{(schemes)} & \sigma & ::= & \tau \mid \forall t_1, \cdots, t_n.\tau \\
\text{(values)} & v & ::= & x \mid i \mid f \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \lambda x{:}\tau.\, e \mid \Lambda t_1, \cdots, t_n.e \\
\text{(expressions)} & e & ::= & x \mid i \mid f \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \\
& & & \mathtt{eq}(e_1, e_2) \mid \mathtt{if0}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\, e_3 \mid \mathtt{def}\, x{:}\sigma = v\ \mathtt{in}\ e_2 \mid \\
& & & v[\tau_1, \cdots, \tau_n]
\end{array}
$$

Figure 2.1: Syntax of Mini-ML

they cannot be placed in data structures, passed as arguments to functions, or returned as the result of a function.

I use `def` instead of `let` because I reserve `let` as an abbreviation. In particular, I use `let` $x{:}\tau = e_1$ `in` e_2 as an abbreviation for $(\lambda x{:}\tau.\, e_2)\ e_1$.

Following conventional formulations of λ-calculus based languages [15], I consider the variable x in $\lambda x{:}\tau.\, e$ to be *bound* within the body of the function e. Likewise, I consider x to be bound within e in the expression `def` $x{:}\sigma = v$ `in` e, and the type variables $t_1, \cdots, t_n$ to be bound within the body of the expression $\Lambda t_1, \cdots, t_n.e$. Likewise, $t_1, \cdots, t_n$ are bound within τ for the polytype $\forall t_1, \cdots, t_n.\tau$. If a variable is not bound in an expression/type, it is said to be *free*. I consider expressions/types to be equivalent modulo α-conversion (i.e., systematic renaming) of the bound variables.

Finally, I write $\{e'/x\}e$ to denote capture avoiding substitution of the closed expression e' for the variable x in the expression e. Likewise, I write $\{\tau/t\}\sigma$ to denote capture-avoiding substitution of the monotype τ for t within the polytype σ.

2.1 Dynamic Semantics of Mini-ML

I describe evaluation of Mini-ML programs using a *contextual* rewriting semantics in the style of Felleisen and Hieb [41]. This kind of semantics describes evaluation as an abstract machine whose states are expressions and whose steps are functions, or more generally, relations between expressions. The final state of this abstract machine is a closed value. Each step of the abstract machine proceeds according to a simple algorithm: We break the current expression into an *evaluation context*, E, and an *instruction* expression, I. The evaluation context is an expression with a "hole" ($[\,]$) in the place of some sub-expression. The original expression, e, is formed by replacing the hole in the context with

$$
\begin{array}{llll}
\text{(contexts)} & E & ::= & [\,] \mid E_1\, e_2 \mid v_1\, E_2 \mid \langle E_1, e_2 \rangle \mid \langle v_1, E_2 \rangle \mid \pi_i\, E \mid \\
& & & \texttt{eq}(E_1, e_2) \mid \texttt{eq}(v_1, E_2) \mid \texttt{if0}\ E_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
\\
\text{(instructions)} & I & ::= & \texttt{eq}(v_1, v_2) \mid \pi_i\, \langle v_1, v_2 \rangle \mid \texttt{if0}\ i\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid (\lambda x{:}\tau.\, e)\, v \mid \\
& & & \texttt{def}\ x{:}\sigma = v\ \texttt{in}\ e \mid (\Lambda t_1, \cdots, t_n.e)\, [\tau_1, \cdots, \tau_n]
\end{array}
$$

Figure 2.2: Contexts and Instructions of Mini-ML

the instruction expression, denoted $e = E[I]$. Roughly speaking, the evaluation context corresponds to the control state (or "stack") of a conventional computer whereas the instruction corresponds to the registers and program counter[1]. We replace the instruction expression with a result expression R within the hole of the context to form a new expression $e' = E[R]$. This new expression serves as the next expression to process in the evaluation sequence.

The expression contexts and instructions of Mini-ML are given in Figure 2.2 and the rewriting rules are given in Figure 2.3. The form of the evaluation contexts reflects the fact that Mini-ML evaluates expressions in a left-to-right, inner-most to outer-most order. Furthermore, the context $v_1\, E_2$ shows that Mini-ML is an *eager* (as opposed to *lazy*) language with respect to function application, because evaluation proceeds on the argument before applying the function to it. Similarly, the contexts for data structures, namely pairs, show that these data structures are eager with respect to their components.

A `def` instruction is evaluated by substituting the value v for all occurrences of the variable x within the scope of the `def`, e. Application of a Λ-expression to a set of monotypes is evaluated by substituting the monotypes for the bound type variables within the body of the abstraction.

Rewriting a primitive instruction is fairly straightforward with the exception of the equality operation. In particular, the rewriting rule for equality must select the appropriate function (e.g., $=_{\mathsf{int}}$ versus $=_{\mathsf{float}}$) according to the syntactic class of the values given to the operation as arguments.

Evaluation does not proceed into the branches of an `if0` construct. Instead, the first component is evaluated and then one of the arms is selected according to the resulting value. When rewriting an application, $(\lambda x{:}\tau.\, e)\, v$, we first substitute the value v for the free occurrences of the variable x within the body of the function e. Likewise, when rewriting a `def` construct, we substitute the value v for the variable x within the body of the `def`.

[1]The relationship between contexts and instructions, and a stack and registers is made explicit in Chapter 7.

$$
\begin{array}{lr}
E[\mathtt{eq}(i_1, i_2)] \longmapsto E[1] & (i_1 =_{\mathsf{int}} i_2) \\
E[\mathtt{eq}(i_1, i_2)] \longmapsto E[0] & (i_1 \neq_{\mathsf{int}} i_2) \\
E[\mathtt{eq}(f_1, f_2)] \longmapsto E[1] & (f_1 =_{\mathsf{float}} f_2) \\
E[\mathtt{eq}(f_1, f_2)] \longmapsto E[0] & (f_1 \neq_{\mathsf{float}} f_2) \\
E[\mathtt{eq}(\langle v_1, v_2\rangle, \langle v_1', v_2'\rangle)] \longmapsto E[\mathtt{if0}\ \mathtt{eq}(v_1, v_1')\ \mathtt{then}\ 0\ \mathtt{else}\, \mathtt{eq}(v_2, v_2')] & \\
E[\mathtt{eq}(\lambda x_1{:}\tau_1.\, e_1, \lambda x_2{:}\tau_2.\, e_2)] \longmapsto E[0] & \\
E[\mathtt{if0}\ 0\ \mathtt{then}\ e_2\ \mathtt{else}\, e_3] \longmapsto E[e_2] & \\
E[\mathtt{if0}\ i\ \mathtt{then}\ e_2\ \mathtt{else}\, e_3] \longmapsto E[e_3] & (i \neq 0) \\
E[\pi_i\, \langle v_1, v_2\rangle] \longmapsto E[v_i] & (i = 1, 2) \\
E[(\lambda x : \tau.\, e)\, v] \longmapsto E[\{v/x\}e] & \\
E[\mathtt{def}\, x{:}\sigma = v\ \mathtt{in}\ e] \longmapsto E[\{v/x\}e] & \\
E[(\Lambda t_1, \cdots, t_n.e)\, [\tau_1, \cdots, \tau_n]] \longmapsto E[\{\tau_1/t_1, \cdots, \tau_n/t_n\}e] &
\end{array}
$$

Figure 2.3: Contextual Dynamic Semantics for Mini-ML

Formally, I consider the rewriting rules to be relations between programs. I consider evaluation to be the least relation formed by taking the reflexive, transitive closure of these rules, denoted by $\longmapsto^*$. I define $e \Downarrow v$ to mean that $e \longmapsto^* v$, and $e \Uparrow$ to mean that there exists an infinite sequence, $e \longmapsto e_1 \longmapsto e_2 \longmapsto \cdots$.

2.2 Static Semantics of Mini-ML

I formulate the static semantics for Mini-ML as a deductive system allowing us to derive judgments of the form $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash v : \sigma$. The first judgment means that under the assumptions of Δ and Γ, the expression e can be assigned the monotype τ. Similarly, the second judgment asserts that the value v can be assigned the type scheme σ under the assumptions of Δ and Γ. Both judgments' assumptions include a set of type variables (Δ) and a type assignment (Γ). The type assignment maps term variables to type schemes, written $\{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\}$. At most one type scheme is assigned to any variable in an assignment. Therefore, we can think of Γ as a partial function that maps variables to types. I use the notation $\Gamma \uplus \{x{:}\tau\}$ to denote the type assignment obtained by extending Γ so that it maps x to τ, under the requirement that x does not already occur in the domain of Γ.

I assume that the free type variables of the range of Γ, the free type variables of e and v, and the free type variables of τ are contained in Δ. Hence, Δ tracks the set of type variables that are in scope for the expression, value, or type. Similarly, the domain of Γ contains the set of free variables of e and v and thus tracks the set of term variables that are in scope for the expression or value. I write $\Delta \vdash \sigma$ and $\Delta \vdash \Gamma$ to assert that σ and Γ are well-formed with respect to Δ.

The axioms and inference rules that allow us to derive these judgments are given in Figure 2.4. Most of the rules are standard, but a few deserve some explanation: If the assumptions map x to the scheme σ, then we can conclude that x has the type σ. Note that x can be viewed as a value or an expression and σ could be a monotype (τ).

The **eq** rule requires that both arguments have the same type. For now, I make no restriction to "equality types" (i.e., types not containing an arrow) as in SML[2]. The dynamic semantics simply maps equality of two functional values to 0.

The most interesting rules are **def**, **tapp**, and **tabs**. The **def** rule allows us to bind a polymorphic value v to some variable within a closed scope e. If we assign the value a quantified type, then we can only use the variable in another `def` binding or type application. Consequently polymorphic objects are "second-class". The **tapp** rule allows us to instantiate a polymorphic value of type $\forall t_1, \cdots, t_n.\tau$, with types $\tau_1, \cdots, \tau_n$. The resulting expression has the monotype formed by replacing t_i with τ_i in τ. Finally, the **tabs** rule assigns the scheme $\forall t_1, \cdots, t_n.\tau$ to the type abstraction $\Lambda t_1, \cdots, t_n.e$ if, adding $t_1, \cdots, t_n$ to the assumptions in Δ, we can conclude that e has type τ. Note that the notation $\Delta \uplus \{t_1, \cdots, t_n\}$ precludes the t_i from occurring in Δ.

I write $\vdash e : \sigma$ if $\emptyset; \emptyset \vdash e : \sigma$ is derivable from these axioms and inference rules. The following lemmas summarize the key properties of the static semantics for Mini-ML.

Lemma 2.2.1 (Type Substitution) *If* $\Delta \uplus \{t\}; \Gamma \vdash e : \sigma$ *and* $\Delta \vdash \tau$, *then* $\Delta; \{\tau/t\}(\Gamma) \vdash \{\tau/t\}(e) : \{\tau/t\}(\sigma)$.

Proof (sketch): By induction on the derivation of $\Delta \uplus \{t\}; \Gamma \vdash e : \sigma$. □

Lemma 2.2.2 (Term Substitution) *If* $\Delta; \Gamma \uplus \{x{:}\sigma'\} \vdash e : \sigma$ *and* $\Delta; \Gamma \vdash e' : \sigma'$, *then* $\Delta; \Gamma \vdash \{e'/x\}e : \sigma$.

Proof (sketch): By induction on $\Delta; \Gamma \uplus \{x{:}\sigma'\} \vdash e : \sigma$. Simply replace all occurrences of the **var** rule used to assign x the type σ' with the derivation of $\Delta; \Gamma \vdash e' : \sigma'$. □

Lemma 2.2.3 (Canonical Forms) *Suppose* $\vdash e : \sigma$. *Then if* σ *is:*

- int, *then* v *is some integer* i.
- float, *then* v *is some floating-point value* f.
- unit, *then* v *is* $\langle\rangle$.

[2] I address the issue of equality types in Section 5.4.2.

$$(\textbf{var})\ \ \Delta;\Gamma \uplus \{x{:}\sigma\} \vdash x : \sigma \qquad (\textbf{int})\ \ \Delta;\Gamma \vdash i : \mathsf{int} \qquad (\textbf{float})\ \ \Delta;\Gamma \vdash f : \mathsf{float}$$

$$(\textbf{eq})\ \ \frac{\Delta;\Gamma \vdash e_1 : \tau \quad \Delta;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{eq}(e_1, e_2) : \mathsf{int}}$$

$$(\textbf{if0})\ \ \frac{\Delta;\Gamma \vdash e_1 : \mathsf{int} \quad \Delta;\Gamma \vdash e_2 : \tau \quad \Delta;\Gamma \vdash e_3 : \tau}{\Delta;\Gamma \vdash \mathtt{if0}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

$$(\textbf{unit})\ \ \Delta;\Gamma \vdash \langle\rangle : \mathsf{unit} \qquad (\textbf{pair})\ \ \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \langle \tau_1 \times \tau_2\rangle}$$

$$(\textbf{proj})\ \ \frac{\Delta;\Gamma \vdash e : \langle \tau_1 \times \tau_2\rangle}{\Delta;\Gamma \vdash \pi_i\, e : \tau_i} \quad (i = 1, 2)$$

$$(\textbf{abs})\ \ \frac{\Delta;\Gamma \uplus \{x{:}\tau_1\} \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad (\textbf{app})\ \ \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$(\textbf{def})\ \ \frac{\Delta;\Gamma \vdash v : \sigma \quad \Delta;\Gamma \uplus \{x{:}\sigma\} \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{def}\ x{:}\sigma = v\ \mathtt{in}\ e : \tau}$$

$$(\textbf{tapp})\ \ \frac{\begin{array}{c}\Delta \vdash \tau_1 \quad \cdots \quad \Delta \vdash \tau_n \\ \Delta;\Gamma \vdash v : \forall t_1, \cdots, t_n.\tau\end{array}}{\Delta \vdash v[\tau_1, \cdots, \tau_n] : \{\tau_1/t_1, \cdots, \tau_n/t_n\}\tau}$$

$$(\textbf{tabs})\ \ \frac{\Delta \uplus \{t_1, \cdots, t_n\};\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda t_1, \cdots, t_n.e : \forall t_1, \cdots, t_n.\tau}$$

Figure 2.4: Static Semantics for Mini-ML

- $\langle \tau_1 \times \tau_2 \rangle$, *then* v *is* $\langle v_1, v_2 \rangle$, *for some* v_1 *and* v_2.
- $\tau_1 \to \tau_2$, *then* v *is* $\lambda x{:}\tau_1.\, e$, *for some* x *and* e.
- $\forall t_1, \cdots, t_n.\tau$, *then* v *is* $\Lambda t_1, \cdots, t_n.e$, *for some* e.

Proof: By an examination of the typing rules. □

Lemma 2.2.4 (Unique Decomposition) *If* $\vdash e : \sigma$, *then either* e *is a value* v *or else there exists a unique* E, e', *and* σ' *such that* $e = E[e']$ *and* $\vdash e' : \sigma'$. *Furthermore, for all* e'' *such that* $\vdash e'' : \sigma'$, $\vdash E[e''] : \sigma$.

Proof (sketch): By induction on the derivation of $\vdash e : \sigma$. Suppose e is not a value. There are seven cases to consider. I give one of the cases here. The other cases follow in a similar fashion.

case: e is $e_1\, e_2$ for some e_1 and e_2. Then a derivation of $\vdash e : \sigma$ must end with a use of the **app** rule. Hence, there exists τ_1 and τ_2 such that $\vdash e_1 : \tau_1 \to \tau_2$, $\vdash e_2 : \tau_1$ and $\sigma = \tau_2$. By induction, either e_1 is a value or else there exists unique E_1, e_1', and σ_1' such that $e_1 = E_1[e_1']$ and $\vdash e_1' : \sigma_1'$. If e_1 is not a value, then we take $E = E_1\, e_2$, $e' = e_1'$, and $\sigma' = \sigma_1'$. Otherwise, $e_1 = v_1$ for some v_1. By induction, either e_2 is a value or else there exists unique E_2, e_2' and σ_2' such that $e_2 = E_2[e_2']$ and $\vdash e_2' : \sigma_2'$. If e_2 is not a value, then $E = v_1\, E_2$, $e' = e_2'$, and $\sigma' = \sigma_2'$. Otherwise, $e_2 = v_2$ for some v_2. Thus, $E = [\,]$, $e' = v_1\, v_2$ and $\sigma' = \sigma$. □

Lemma 2.2.5 (Preservation) *If* $\vdash e : \sigma$ *and* $e \longmapsto e'$, *then* $\vdash e' : \sigma$.

Proof: By Unique Decomposition and the fact that $e \longmapsto e'$, there exists a unique E, I, and e'' such that $e = E[I]$, $I \longmapsto e''$, and $e' = E[e'']$. Furthermore, there exists a σ' such that $\vdash I : \sigma'$ and for all e''' such that $\vdash e''' : \sigma'$, $\vdash E[e'''] : \sigma$. Hence, it suffices to show that regardless of I and e'', $\vdash e'' : \sigma'$. There are six cases to consider:

case: If $I = \mathtt{eq}(v_1, v_2)$ then a derivation of $\vdash I : \sigma'$ must end with an application of the **eq** rule. Hence, $\sigma' = \mathtt{int}$ and there exists a τ such that $\vdash v_1 : \tau$ and $\vdash v_2 : \tau$. If v_1 and v_2 are integers, floats, or λ-abstractions, then $I \longmapsto i$ for some i and $\vdash i : \sigma'$. If v_1 and v_2 are pairs, $\langle v_a, v_b \rangle$ and $\langle v_a', v_b' \rangle$, then $I \longmapsto \mathtt{if0}\ \mathtt{eq}(v_a, v_a')\ \mathtt{then}\ 0\ \mathtt{else}\, \mathtt{eq}(v_b, v_b')$. By examination of the typing rules, τ must be of the form $\langle \tau_a \times \tau_b \rangle$. Since $\vdash v_1 : \langle \tau_a \times \tau_b \rangle$ and $\vdash v_2 : \langle \tau_a \times \tau_b \rangle$, derivations of these facts must end with a use of the **pair** rule. Hence, $\vdash v_a : \tau_a$, $\vdash v_b : \tau_b$, $\vdash v_a' : \tau_a$, and $\vdash v_b' : \tau_b$. By the **eq** rule, $\vdash \mathtt{eq}(v_a, v_a') : \mathtt{int}$ and $\vdash \mathtt{eq}(v_b, v_b') : \mathtt{int}$. Thus, by the **if0** rule, $\vdash \mathtt{if0}\ \mathtt{eq}(v_a, v_a')\ \mathtt{then}\ 0\ \mathtt{else}\, \mathtt{eq}(v_b, v_b') : \mathtt{int}$. Hence, $\vdash e'' : \sigma'$.

case: If $I = \texttt{if0}\ i\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, then a derivation of $\vdash I : \sigma'$ must end with an application of the **if0** rule. Hence, there exists a τ such that $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$ and $\sigma' = \tau$. If i is 0, then $I \longmapsto e_1$ else $I \longmapsto e_2$. Regardless, the resulting expression has type σ'.

case: If $I = \pi_i\, v$ for $i = 1, 2$, then a derivation of $\vdash I : \sigma'$ must end with an application of the **proj** rule. Hence, there exists τ_1 and τ_2 such that $\vdash v : \langle\tau_1 \times \tau_2\rangle$ and $\sigma' = \tau_i$. By Canonical Forms, v must be of the form $\langle v_1, v_2\rangle$ and $I \longmapsto v_i$. A derivation of $\vdash \langle v_1, v_2\rangle : \langle\tau_1 \times \tau_2\rangle$ must end with the **pair** rule, hence $\vdash v_i : \tau_i$ and $\vdash e'' : \sigma'$.

case: If $I = (\lambda x{:}\tau_1.\, e_1)\, v$ then $e'' = \{v/x\}e_1$. A derivation of $\vdash I : \sigma'$ must end with a use of the **app** rule. Hence, $\sigma' = \tau_2$ for some τ_2 and $\vdash \lambda x{:}\tau_1.\, e_1 : \tau_1 \to \tau_2$ and $\vdash v : \tau_1$. By Term Substitution, $\vdash \{v/x\}e_1 : \tau_2$. Hence, $\vdash e'' : \sigma'$.

case: If $I = (\Lambda t_1, \cdots, t_n.e_1)\,[\tau_1, \cdots, \tau_n]$ then $e'' = \{\tau_1/t_1, \cdots, \tau_n/t_n\}e_1$. A derivation of $\vdash I : \sigma'$ must end with a use of the **tapp** rule. Hence, $\sigma' = \{\tau_1/t_1, \cdots, \tau_n/t_n\}\tau$ for some τ such that $\vdash \Lambda t_1, \cdots, t_n.e_1 : \forall t_1, \cdots, t_n.\tau$. By Type Substitution, $\vdash \{\tau_1/t_1, \cdots, \tau_n/t_n\}e_1 : \{\tau_1/t_1, \cdots, \tau_n/t_n\}\tau$. Thus, $\vdash e'' : \sigma'$. □

Lemma 2.2.6 (Progress) *If $\vdash e : \sigma$, then either e is a value or else there exists some e' such that $e \longmapsto e'$.*

Proof: If e is not a value, then by Unique Decomposition, there exists an E, e_1, and σ' such that $e = E[e_1]$ and $\vdash e_1 : \sigma'$. I argue that e_1 must be an instruction and hence, there is an e_2 such that $e_1 \longmapsto e_2$ and thus $E[e_1] \longmapsto E[e_2]$. There are five cases to consider, where e_1 could possibly be stuck.

case: If e_1 is of the form $e_a\, e_b$, then e_a and e_b must both be values, else by Unique Decomposition, e_1 can be broken into a nested evaluation context and expression. Thus, $e_1 = v_1\, v_2$ for some v_1 and v_2. Since $\vdash v_1\, v_2 : \sigma'$, the derivation must end in an application of the **app** rule. Thus, there exists a τ' and τ such that $\vdash v_1 : \tau' \to \tau$ and $\vdash v_2 : \tau'$ and $\sigma' = \tau$. Since $\vdash v_1 : \tau' \to \tau$, v_1 is closed, by Canonical Forms, v_1 must be of the form $\lambda x{:}\tau'.\, e''$ for some x and e''. Thus, $e_1 \longmapsto \{v_2/x\}e''$.

case: If e_1 is of the form $\pi_i\, e_1$ for $i = 1, 2$, then e_1 must be a value v_1, else by Unique Decomposition, e_1 can be broken into a nested evaluation context and expression. Thus, $e_1 = v$ for some v. Since $\vdash \pi_i\, v : \sigma'$, by an examination of the typing rules, a derivation of this fact must end with a use of the **proj** rule. Hence, $\vdash v : \tau_1 \times \tau_2$ for some τ_1 and τ_2 such that $\tau_i = \sigma'$. By Canonical Forms, there exists two values v_1 and v_2 such that $v = \langle v_1, v_2\rangle$. Hence, $e_1 \longmapsto v_i$.

case: If e_1 is of the form $\texttt{eq}(e_a, e_b)$ then e_a and e_b must be values, v_1 and v_2. Since $\vdash e_1 : \sigma'$, a derivation of this fact must end with the **eq** rule. Hence, σ' is int and there exists a τ such that $\vdash v_1 : \tau$ and $\vdash v_2 : \tau$. By Canonical Forms, v_1 and v_2 are both either integers, floats, pairs, or functions. Hence, $e_1 \longmapsto i$ for some i.

case: If e_1 is of the form `if0` e_a `then` e_b `else` e_c, then e_a must be a value v. Since $\vdash e_1 : \sigma'$, a derivation of this fact must end with a use of the **if0** rule. Hence, $\vdash v : \mathsf{int}$. By canonical forms, v is some integer i. If i is 0, then $e_1 \longmapsto e_b$ else $e_1 \longmapsto e_c$.
case: If e_1 is of the form $v\,[\tau_1, \cdots, \tau_n]$, then since $\vdash e_1 : \sigma'$, a derivation of this fact must end with a use of the **tapp** rule. Hence, there exists some $\forall t_1, \cdots, t_n.\tau$ such that $\vdash v : \forall t_1, \cdots, t_n.\tau$, where $\sigma' = \{\tau_1/t_1, \cdots, \tau_n/t_n\}\tau$. By Canonical Forms, v must be $\Lambda t_1, \cdots, t_n.e''$ for some e''. Hence, $e_1 \longmapsto \{\tau_1/t_1, \cdots, \tau_n/t_n\}e''$. □

This last lemma implies that well-typed Mini-ML expressions cannot "get stuck" during evaluation. From a practical standpoint, this means that it is impossible to have a well-typed program that attempts to apply a non-function to some arguments, or to project a component from a non-tuple. Therefore, any implementation that accurately reflects the dynamic semantics will never "dump core" when given a well-typed program.

Corollary 2.2.7 (Soundness) *If $\vdash e : \sigma$, then either $e \Uparrow$ or else there exists some v such that $e \Downarrow v$ and $\vdash v : \sigma$.*

Proof: By induction on the number of rewriting steps, if $e \longmapsto^* e'$, then by Preservation, $\vdash e' : \sigma$ and by Progress, either e' is a value or else there exists an e'' such that $e' \longmapsto e''$. Therefore, either there exists an infinite sequence, $e \longmapsto^* e' \longmapsto e_1 \longmapsto e_2 \longmapsto \cdots$, or else $e \Downarrow v$ and $\vdash v : \sigma$. □

Chapter 3

A Calculus of Dynamic Type Dispatch

I argued in Chapter 1 that compiling a polymorphic language without sacrificing control over data representations or the ability to compile modules separately requires an intermediate language that supports dynamic type dispatch. In this chapter, I present a core calculus called λ_i^{ML} that provides a formal foundation for dynamic type dispatch. In subsequent chapters, I derive intermediate languages based on this formal calculus and show how to compile Mini-ML to these lower-level languages, taking advantage of dynamic type dispatch to implement various language features.

3.1 Syntax of λ_i^{ML}

λ_i^{ML} is based on λ^{ML} [94], a predicative variant of the Girard-Reynolds polymorphic calculus, F_ω [47, 46, 106]. The essential departure from the impredicative systems of Girard and Reynolds is that, as in Mini-ML, there is a distinction made between monotypes (types without a quantifier) and polytypes, and type variables are only allowed to range over monotypes. Such a calculus is more than sufficient for the interpretation of ML-style polymorphism[1] and makes arguments based on logical relations easier than an impredicative calculus. The language λ_i^{ML} extends λ^{ML} with *intensional* (or *structural* [52]) polymorphism, which allows non-parametric functions to be defined via intensional analysis of types.

The four syntactic classes of λ_i^{ML} are given in Figure 3.1. The expressions of the language are described by types. Types include int, function types, explicitly injected constructors ($T(\mu)$) and polymorphic types ($\forall t{::}\kappa.\sigma$). Types that do not include a quantifier are called monotypes, whereas types that do include a quantifier are called polytypes.

[1]See Harper and Mitchell [94] for further discussion of this point.

The language easily extends to float, products, and inductively generated types like lists; I omit these here to simplify the formal treatment of the calculus.

The constructors of λ_i^{ML} form a language that is isomorphic to a simply-typed λ-calculus extended with a single, inductively defined base type (such as lists or trees) and an induction elimination form (such as fold). In this case, the inductively defined base type is given by the set of constructor values which are generated as follows:

$$\tau \quad ::= \quad \mathsf{Int} \mid \mathsf{Arrow}(\tau_1, \tau_2).$$

Each constructor value τ *names* a monotype σ. In particular, Int is a constructor value that names the type int. If τ_1 names the type σ_1 and τ_2 names the type σ_2, then $\mathsf{Arrow}(\tau_1, \tau_2)$ names the type $\sigma_1 \to \sigma_2$.

To distinguish expression-level types from constructor-level types, I call the latter *kinds* (κ). Closed constructors of kind Ω compute constructor values. If μ computes the constructor value τ, and τ names the monotype σ, then I use the explicit injection $T(\mu)$ to denote the monotype σ. The precise relationship between constructors and monotypes is axiomatized in Section 3.3.

As in standard polymorphic calculi, constructor abstractions ($\Lambda t{::}\kappa.e$) let us define functions from constructors to terms. Unlike languages based on the Hindley-Milner type system including Mini-ML, SML, and Haskell, I do not restrict constructor abstractions to a "second-class" status. This is reflected in the types of the language, because there is no prenex-quantifier restriction. Hence, constructor abstractions can be placed in data structures, passed as arguments, or returned from functions.

The Typerec and `typerec` forms give us the ability to define both constructors and terms by structural induction on monotypes. The Typerec and `typerec` forms may be thought of as eliminatory forms for the kind Ω at the constructor and term level respectively. The introductory forms are the constructors of kind Ω; there are no introductory forms at the term level in order to preserve the phase distinction [25, 60]. In effect, Typerec and `typerec` let us *fold* some computation over a monotype. Limiting the computation to a fold, instead of some general recursion, ensures that the computation terminates — a crucial property at the constructor level. However, many useful operations, including pattern matching, iterators, maps, and reductions can be coded using folds.

I consider λ to bind the type variable t within a constructor function, $\lambda t{::}\kappa.\, e$. I also consider $\forall t{::}\kappa.\sigma$ to bind t within the scope of σ. I consider λ to bind the expression variable x within an expression function, $\lambda x{:}\sigma.\, e$. I consider Λ to bind the type variable t within a constructor abstraction $\Lambda t{::}\kappa.e$. Finally, I consider t to be bound in σ for the "$[t.\sigma]$" portion of a `typerec` expression. This type scheme on `typerec`s is needed to make the language explicitly typed. As usual, I consider constructors, types, and expressions to be equivalent modulo α-conversion of bound variables.

$$
\begin{array}{llll}
\text{(kinds)} & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2 \\[1ex]
\text{(constructors)} & \mu & ::= & t \mid \mathsf{Int} \mid \mathsf{Arrow}(\mu_1, \mu_2) \mid \lambda t{::}\kappa.\, \mu \mid \mu_1\, \mu_2 \mid \\
 & & & \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_{\text{int}}; \mu_{\text{arrow}}) \\[1ex]
\text{(types)} & \sigma & ::= & T(\mu) \mid \mathsf{int} \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma \\[1ex]
\text{(expressions)} & e & ::= & x \mid i \mid \lambda x{:}\sigma.\, e \mid \Lambda t{::}\kappa.e \mid e_1\, e_2 \mid e[\mu] \mid \\
 & & & \mathtt{typerec}\ \mu\ \mathtt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}})
\end{array}
$$

Figure 3.1: Syntax of λ_i^{ML}

$$
\begin{array}{llll}
\text{(values)} & u & ::= & \mathsf{Int} \mid \mathsf{Arrow}(u, u) \mid \lambda t{::}\kappa.\, \mu \\[1ex]
\text{(contexts)} & U & ::= & [\,] \mid U\, \mu \mid u\, U \mid \mathsf{Typerec}\ U\ \mathsf{of}\ (\mu_{\text{int}}; \mu_{\text{arrow}}) \\[1ex]
\text{(instructions)} & J & ::= & (\lambda t{::}\kappa.\, \mu_1)\, u \mid \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_{\text{int}}; \mu_{\text{arrow}}) \mid \\
 & & & \mathsf{Typerec}\ \mathsf{Arrow}(u_1, u_2)\ \mathsf{of}\ (\mu_{\text{int}}; \mu_{\text{arrow}})
\end{array}
$$

Figure 3.2: Values, Contexts, and Instructions of Constructors

3.2 Dynamic Semantics of λ_i^{ML}

The dynamic semantics for λ_i^{ML} consists of a set of rewriting rules for both constructors and expressions. I use a contextual semantics to describe evaluation at both levels.

The values, evaluation contexts, and instructions for constructors are given in Figure 3.2. I choose to evaluate constructors in a call-by-value fashion, though either call-by-name or call-by-need would also be appropriate. Therefore, the values of constructors consist of variables, functions, Int, or Arrow constructors with value components. The evaluation contexts of constructors consist of a hole, an application with a hole somewhere in the function position, an application of a value to constructor with a hole in the argument position, or a Typerec with a hole somewhere in the argument. The instructions consist of an application of a function to a value, or a Typerec where the the argument constructor is either Int or Arrow.

The rewriting rules for constructors are given in Figure 3.3. The rule for function

$$U[(\lambda t{::}\kappa.\,\mu_1)\,u_2] \longmapsto U[\{u_2/t\}\mu_1]$$

$$U[\textsf{Typerec Int of } (\mu_{\text{int}};\mu_{\text{arrow}})] \longmapsto U[\mu_{\text{int}}]$$

$$\begin{array}{l} U[\textsf{Typerec Arrow}(u_1,u_2) \textsf{ of } (\mu_{\text{int}};\mu_{\text{arrow}})] \longmapsto \\ \quad U[\mu_{\text{arrow}}\,u_1\,u_2\,(\textsf{Typerec } u_1 \textsf{ of } (\mu_{\text{int}};\mu_{\text{arrow}}))\,(\textsf{Typerec } u_2 \textsf{ of } (\mu_{\text{int}};\mu_{\text{arrow}}))] \end{array}$$

Figure 3.3: Rewriting Rules for Constructors

application is straightforward. The rules for Typerec select the appropriate clause according to the head component of the argument constructor. Thus, μ_{int} is chosen if the argument is Int, while μ_{arrow} is chosen if the argument is Arrow(u_1, u_2). If the argument constructor has components, then we pass these components as arguments to the clause. We also pass the the "unrolling" of the Typerec on these components. For instance, if the argument constructor is Arrow(u_1, u_2), then we pass u_1, u_2, and the same Typerec applied to u_1 and u_2 to the μ_{arrow} clause. In this fashion, the Typerec is folded across the components of a constructor.

The values, evaluation contexts, and instructions for the expressions of λ_i^{ML} are given in Figure 3.4. The evaluation contexts and values show that λ_i^{ML} expressions are evaluated in a standard call-by-value fashion. As at the constructor level, I choose to evaluate constructor application eagerly. Hence, a constructor is reduced to a constructor value before it is substituted for the Λ-bound type variable of a constructor abstraction. Evaluation of an expression-level `typerec` is similar to the evaluation of a constructor-level Typerec. First, the argument is evaluated and then, the appropriate clause, either e_{int} or e_{arrow} is chosen according to this component. Any nested constructor components are passed as constructor arguments to the clause as well as the "unrolling" of the `typerec` on these components. Hence, evaluation of a `typerec` applied to the constructor Arrow(u_1, u_2) selects the e_{arrow} clause, passes it u_1 and u_2 as constructor arguments and the same `typerec` applied to u_1 and u_2 as value arguments.

$$
\begin{array}{llll}
\text{(values)} & v & ::= & i \mid \lambda x{:}\sigma.\, e \mid \Lambda t{::}\kappa.e \\[1em]
\text{(contexts)} & E & ::= & [] \mid E_1\, e_2 \mid v_1\, E_2 \mid E[\mu] \\[1em]
\text{(instructions)} & I & ::= & (\lambda x{:}\sigma.\, e)\, v \mid (\Lambda t{::}\kappa.e)[U[J]] \mid (\Lambda t{::}\kappa.e)[u] \mid \\
 & & & \texttt{typerec}\ U[J]\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}}) \mid \\
 & & & \texttt{typerec}\ \mathsf{Int}\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}}) \mid \\
 & & & \texttt{typerec}\ \mathsf{Arrow}(u_1, u_2)\ \texttt{of}\ [t.\sigma](e_{\text{int}};; e_{\text{arrow}})
\end{array}
$$

Figure 3.4: Values, Contexts, and Instructions of Expressions

$$E[(\lambda x{:}\sigma.\, e)\, v] \longmapsto E[\{v/x\}e]$$

$$E[(\Lambda t{::}\kappa.e)[U[J]]] \longmapsto E[(\Lambda t{::}\kappa.e)[U[\mu]]] \quad \text{when} \quad U[J] \longmapsto U[\mu]$$

$$E[(\Lambda t{::}\kappa.e)[u]] \longmapsto E[\{u/t\}e]$$

$$
\begin{array}{l}
E[\texttt{typerec}\ U[J]\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}})] \longmapsto \\
\quad E[\texttt{typerec}\ U[\mu]\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}})] \quad \text{when} \quad U[J] \longmapsto U[\mu]
\end{array}
$$

$$E[\texttt{typerec}\ \mathsf{Int}\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}})] \longmapsto E[e_{\text{int}}]$$

$$
\begin{array}{l}
E[\texttt{typerec}\ \mathsf{Arrow}(u_1, u_2)\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}})] \longmapsto \\
\quad E[e_{\text{arrow}}\, [u_1]\, [u_2]\, (\texttt{typerec}\ u_1\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}})) \\
\qquad (\texttt{typerec}\ u_2\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}}))]
\end{array}
$$

Figure 3.5: Rewriting Rules for Expressions

$$(\textbf{var})\ \ \Delta \uplus \{t{::}\kappa\} \vdash t :: \kappa \qquad\qquad (\textbf{int})\ \ \Delta \vdash \mathsf{Int} :: \Omega$$

$$(\textbf{arrow})\ \ \frac{\Delta \vdash \mu_1 :: \Omega \quad \Delta \vdash \mu_2 :: \Omega}{\Delta \vdash \mathsf{Arrow}(\mu_1, \mu_2) :: \Omega}$$

$$(\textbf{fn})\ \ \frac{\Delta \uplus \{t{::}\kappa_1\} \vdash \mu :: \kappa_2}{\Delta \vdash \lambda t{::}\kappa_1.\, \mu :: \kappa_1 \to \kappa_2} \qquad\qquad (\textbf{app})\ \ \frac{\Delta \vdash \mu_1 : \kappa_1 \to \kappa_2 \quad \Delta \vdash \mu_2 : \kappa_1}{\Delta \vdash \mu_1\, \mu_2 : \kappa_2}$$

$$(\textbf{trec})\ \ \frac{\begin{array}{c}\Delta \vdash \mu :: \Omega \quad \Delta \vdash \mu_{\text{int}} :: \kappa \\ \Delta \vdash \mu_{\text{arrow}} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\Delta \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_{\text{int}}; \mu_{\text{arrow}}) :: \kappa}$$

Figure 3.6: Constructor Formation

3.3 Static Semantics of λ_i^{ML}

The static semantics of λ_i^{ML} consists of a collection of rules for deriving judgments of the form

$\Delta \vdash \mu :: \kappa$	μ is a constructor of kind κ
$\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$	μ_1 and μ_2 are equivalent constructors
$\Delta \vdash \sigma$	σ is a valid type
$\Delta \vdash \sigma_1 \equiv \sigma_2$	σ_1 and σ_2 are equivalent types
$\Delta; \Gamma \vdash e : \sigma$	e is a term of type σ,

where Δ is a kind assignment, mapping type variables (t) to kinds (κ), and Γ is a type assignment, mapping term variables (x) to types (σ). These judgments may be derived from the axioms and inference rules of Figures 3.6, 3.7, 3.8, 3.9, and 3.10, respectively.

Constructor formation (see Figure 3.6) is standard with the exception of Typerec. Here, I require that the argument of the Typerec be of kind Ω. When evaluating a Typerec, one of the clauses is chosen according to the value of the argument. Any components are passed as arguments to the clause as well as the unrolling of the Typerec on these components. Therefore, the whole constructor is assigned the kind κ only if μ_{int} has kind κ (since the Int constructor has no components) and μ_{arrow} has kind $\Omega \to \Omega \to \kappa \to \kappa \to \kappa$, (since it has two components).

To type check an expression, we need to be able to tell when two types are equivalent. Since constructors can be injected into types, we need an appropriate notion of constructor equivalence. Therefore, I define *definitional equivalence* [113, 87] via the judgment $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$. Figure 3.7 gives the axioms and inference rules that allows us to derive definitional equivalence. The rules consist of β- and η-conversion, recursion equations governing the Typerec form, and standard rules of equivalence and congruence. In the following chapter, I show that every well-formed constructor μ has a unique normal form, with respect to the obvious notion of reduction derived by orienting these equivalence rules to the right. Furthermore, I show that this reduction relation is confluent, from which it follows that constructor equivalence is decidable [113].

The type formation and equivalence rules can be found in Figures 3.8 and 3.9 respectively. The rules of type equivalence define the interpretation $T(\mu)$ of the constructor μ as a type. For example, $T(\mathsf{Int}) \equiv \mathsf{int}$ and $T(\mathsf{Arrow}(\mu_1, \mu_2)) \equiv T(\mu_1) \to T(\mu_2)$. Thus, T takes us from a constructor that *names* a type to the actual type. The other type equivalence rules make the relation an equivalence and congruence with respect to the type constructs.

The term formation rules may be found in Figure 3.10. Term formation judgments are of the form $\Delta; \Gamma \vdash e : \sigma$. I make the implicit assumption that all free type variables in σ, e, and the range of Γ can be found in the domain of Δ. Hence, Δ provides the set of type variables that are in scope.

The term formation rules resemble the typing rules of Mini-ML (see Figure 2.4) with the exception of the constructor abstraction, application, `typerec` and equivalence rules. Similar to value abstraction, constructor abstraction adds a new type variable of the appropriate kind to the current kind assignment to give a type to the body of the abstraction. Again, the "$\uplus$" notation ensures that the added variable does not already occur in Δ. For a constructor application, $e\,[\mu]$, if e is given a polymorphic type $\forall t{::}\kappa.\sigma$, and μ has kind κ under the current type and kind assignments, then the resulting expression has the type obtained by substituting μ for the free occurrences of t within σ. The equivalence rule ascribes the type σ to an expression of type σ' if σ and σ' are definitionally equivalent.

A `typerec` expression of the form `typerec` μ `of` $[t.\sigma](e_{\mathrm{int}}; e_{\mathrm{arrow}})$ is given the type obtained by substituting the argument constructor μ for t in the type σ if the following conditions hold: First, μ must be a constructor of kind Ω, since only these constructors can be examined via `typerec`. Second, each of the clauses must have a type obtained by replacing the appropriate constructor for t within σ. Furthermore, e_{arrow} must abstract the components of the Arrow constructor as well as the result of unwinding the `typerec` on these components.

3.4 Related Work

There are two traditional interpretations of polymorphism, the *explicit* style (due to Girard [47, 46] and Reynolds [106]), in which types are passed to polymorphic operations, and the *implicit* style (due to Milner [89]), in which types are erased prior to execution.

$$(\beta)\ \frac{\Delta \uplus \{t :: \kappa'\} \vdash \mu_1 :: \kappa \qquad \Delta \vdash \mu_2 :: \kappa'}{\Delta \vdash (\lambda t{::}\kappa'.\, \mu_1)\, \mu_2 \equiv \{\mu_2/t\}\mu_1 :: \kappa}$$

$$(\eta)\ \frac{\Delta \vdash \mu :: \kappa_1 \to \kappa_2}{\Delta \vdash \lambda t{::}\kappa_1.\, (\mu\ t) \equiv \mu :: \kappa_1 \to \kappa_2} \quad (t \notin Dom(\Delta))$$

$$(\textbf{trec-int})\ \frac{\begin{array}{c}\Delta \vdash \mu_{\text{int}} :: \kappa \\ \Delta \vdash \mu_{\text{arrow}} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\Delta \vdash \textsf{Typerec Int of } (\mu_{\text{int}}; \mu_{\text{arrow}}) \equiv \mu_{\text{int}} :: \kappa}$$

$$(\textbf{trec-arrow})\ \frac{\begin{array}{c}\Delta \vdash \mu_1 :: \Omega \quad \Delta \vdash \mu_2 :: \Omega \quad \Delta \vdash \mu_{\text{int}} :: \kappa \\ \Delta \vdash \mu_{\text{arrow}} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\begin{array}{ll}\Delta \vdash & \textsf{Typerec Arrow}(\mu_1, \mu_2) \textsf{ of } (\mu_{\text{int}}; \mu_{\text{arrow}}) \equiv \\ & \quad \mu_{\text{arrow}}\ \mu_1\ \mu_2\ (\textsf{Typerec } \mu_1 \textsf{ of } (\mu_{\text{int}}; \mu_{\text{arrow}})) \\ & \qquad\qquad (\textsf{Typerec } \mu_2 \textsf{ of } (\mu_{\text{int}}; \mu_{\text{arrow}})) :: \kappa\end{array}}$$

$$(\textbf{refl})\ \frac{\Delta \vdash \mu :: \kappa}{\Delta \vdash \mu \equiv \mu :: \kappa} \qquad (\textbf{tran})\ \frac{\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa \quad \Delta \vdash \mu_2 \equiv \mu_3 :: \kappa}{\Delta \vdash \mu_1 \equiv \mu_3 :: \kappa}$$

$$(\textbf{symm})\ \frac{\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa}{\Delta \vdash \mu_2 \equiv \mu_1 :: \kappa} \qquad (\textbf{arrow})\ \frac{\Delta \vdash \mu_1 \equiv \mu_1' :: \Omega \quad \Delta \vdash \mu_2 \equiv \mu_2' :: \Omega}{\Delta \vdash \textsf{Arrow}(\mu_1, \mu_2) \equiv \textsf{Arrow}(\mu_1', \mu_2') :: \Omega}$$

$$(\textbf{fn})\ \frac{\Delta \uplus \{t{::}\kappa_1\} \vdash \mu_1 \equiv \mu_2 :: \kappa_2}{\Delta \vdash \lambda t{::}\kappa_1.\, \mu_1 \equiv \lambda t{::}\kappa_1.\, \mu_2 :: \kappa_1 \to \kappa_2}$$

$$(\textbf{app})\ \frac{\Delta \vdash \mu_1 \equiv \mu_1' :: \kappa_1 \to \kappa_2 \quad \Delta \vdash \mu_2 \equiv \mu_2' :: \kappa_1}{\Delta \vdash \mu_1\, \mu_2 \equiv \mu_1'\, \mu_2' :: \kappa_2}$$

$$(\textbf{trec})\ \frac{\begin{array}{c}\Delta \vdash \mu \equiv \mu' :: \Omega \quad \Delta \vdash \mu_{\text{int}} \equiv \mu_{\text{int}}' :: \kappa \\ \Delta \vdash \mu_{\text{arrow}} \equiv \mu_{\text{arrow}}' :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\Delta \vdash \textsf{Typerec } \mu \textsf{ of } (\mu_{\text{int}}; \mu_{\text{arrow}}) \equiv \textsf{Typerec } \mu' \textsf{ of } (\mu_{\text{int}}'; \mu_{\text{arrow}}') :: \kappa}$$

Figure 3.7: Constructor Equivalence

$$\Delta \vdash \mathsf{int} \qquad \frac{\Delta \vdash \mu :: \Omega}{\Delta \vdash T(\mu)} \qquad \frac{\Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \to \sigma_2} \qquad \frac{\Delta \uplus \{t{::}\kappa\} \vdash \sigma}{\Delta \vdash \forall t{::}\kappa.\, \sigma}$$

Figure 3.8: Type Formation

$$\Delta \vdash T(\mathsf{Int}) \equiv \mathsf{int} \qquad \Delta \vdash T(\mathsf{Arrow}(\mu_1, \mu_2)) \equiv T(\mu_1) \to T(\mu_2)$$

$$\frac{\Delta \vdash \mu \equiv \mu' :: \Omega}{\Delta \vdash T(\mu) \equiv T(\mu')}$$

$$\Delta \vdash \sigma \equiv \sigma \qquad \frac{\Delta \vdash \sigma \equiv \sigma'}{\Delta \vdash \sigma' \equiv \sigma} \qquad \frac{\Delta \vdash \sigma_1 \equiv \sigma_2 \quad \Delta \vdash \sigma_2 \equiv \sigma_3}{\Delta \vdash \sigma_1 \equiv \sigma_3}$$

$$\frac{\Delta \vdash \sigma_1 \equiv \sigma_1' \quad \Delta \vdash \sigma_2 \equiv \sigma_2'}{\Delta \vdash \langle \sigma_1 \times \sigma_2 \rangle \equiv \langle \sigma_1' \times \sigma_2' \rangle} \qquad \frac{\Delta \vdash \sigma_1 \equiv \sigma_1' \quad \Delta \vdash \sigma_2 \equiv \sigma_2'}{\Delta \vdash \sigma_1 \to \sigma_2 \equiv \sigma_1' \to \sigma_2'}$$

$$\frac{\Delta \uplus \{t{::}\kappa\} \vdash \sigma \equiv \sigma'}{\Delta \vdash \forall t{::}\kappa.\sigma \equiv \forall t{::}\kappa.\sigma'}$$

Figure 3.9: Type Equivalence

$$(\textbf{var})\ \ \Delta;\Gamma \uplus \{x{:}\sigma\} \vdash x : \sigma \qquad\qquad (\textbf{int})\ \ \Delta;\Gamma \vdash i : \mathsf{int}$$

$$(\textbf{fn})\ \ \frac{\Delta;\Gamma \uplus \{x{:}\sigma_1\} \vdash e : \sigma_2}{\Delta;\Gamma \vdash \lambda x{:}\sigma_1.\, e : \sigma_1 \to \sigma_2} \qquad (\textbf{app})\ \ \frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \quad \Delta;\Gamma \vdash e_2 : \sigma_1}{\Delta;\Gamma \vdash e_1\, e_2 : \sigma_2}$$

$$(\textbf{tfn})\ \ \frac{\Delta \uplus \{t{::}\kappa\};\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \Lambda t{::}\kappa.\, e : \forall t{::}\kappa.\sigma} \qquad (\textbf{tapp})\ \ \frac{\Delta;\Gamma \vdash e : \forall t{::}\kappa.\sigma \quad \Delta \vdash \mu :: kappa}{\Delta;\Gamma \vdash e[\mu] : \{\mu/t\}\sigma}$$

$$(\textbf{trec})\ \ \frac{\begin{array}{c}\Delta \vdash \mu : \Omega \\ \Delta;\Gamma \vdash e_{\text{int}} : \{\mathsf{Int}/t\}\sigma \\ \Delta;\Gamma \vdash e_{\text{arrow}} : \forall t_1{::}\Omega.\forall t_2{::}\Omega.\{t_1/t\}\sigma \to \{t_2/t\}\sigma \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma\end{array}}{\Delta;\Gamma \vdash \texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma](e_{\text{int}}; e_{\text{arrow}}) : \{\mu/t\}\sigma}$$

$$(\textbf{equiv})\ \ \frac{\Delta;\Gamma \vdash e : \sigma' \qquad \Delta \vdash \sigma' \equiv \sigma :: \Omega}{\Delta;\Gamma \vdash e : \sigma}$$

Figure 3.10: Term Formation

In their study of the type theory of Standard ML [93, 59], Harper and Mitchell argued that an explicitly-typed interpretation of ML polymorphism has better semantic properties and scales more easily to cover a full programming language. Harper and Mitchell formulated a predicative type theory, XML, a theory of dependent types augmented with a universe of small types, adequate for capturing many aspects of SML. This type theory was later refined by Harper, Mitchell, and Moggi [60], and provides the fundamental basis for the type theory of λ_i^{ML}.

The idea of adding an inductively generated ground type (the natural numbers) with an elimination rule like $\mathsf{Typerec}$, to the typed λ-calculus is implicit in Gödel's original "functionals of finite type"[48]. Thus, the constructor language of λ_i^{ML} is fundamentally based on this work. According to Lambek and Scott [79], Marie-France Thibault [117] studied the correspondence between such calculi and cartesian closed categories equipped with "strong" natural number objects. However, the notion of constructor equivalence in λ_i^{ML} corresponds to what Lambek and Scott term a "weak" natural number object.

The idea of adding an inductively generated *universe*, with a term-level elimination rule such as `typerec`, was derived from the universe elimination rules found in NuPrl [32], though the idea was only described in unpublished work of Robert Constable. Harper and I devised the original formulation of λ_i^{ML} [62, 61].

Chapter 4

Typing Properties of λ_i^{ML}

In this chapter, I present proofs of two important properties of λ_i^{ML}: Type checking λ_i^{ML} terms is decidable, and the type system is sound with respect to the operational semantics. Hence, λ_i^{ML} enjoys many of the same semantic properties as more conventional typed calculi.

Readers anxious to see how dynamic type dispatch can be used may wish to skip this chapter and come back to it later.

4.1 Decidability of Type Checking for λ_i^{ML}

If we remove the equivalence rule from the term formation rules, then type checking λ_i^{ML} terms would be entirely syntax-directed. This is due to the type label on λ-abstractions at the expression level, the kind label on λ-abstractions at the constructor level, and the type scheme $[t.\sigma]$ labelling `typerec` expressions. But in the presence of the equivalence rule, we need an alternative method for determining whether an expression may be assigned a type, and if so, what types it may be given.

In this section, I show that every well-formed constructor has a unique normal form with respect to a certain notion of reduction, and that two constructors are definitionally equivalent if and only if they have syntactically identical normal forms, modulo α-conversion. From this notion of normal forms for constructors, it is straightforward to generalize to a normal form for types: Normalize any constructor components of the type, and recursively replace $T(\mathsf{Int})$ with int and $T(\mathsf{Arrow}(\mu_1, \mu_2))$ with $T(\mu_1) \to T(\mu_2)$. From this, it is easy to see that two types are definitionally equivalent iff their normal forms are syntactically equivalent, modulo α-conversion.

With normal forms for types, we can formulate a different proof system for the well-formedness of λ_i^{ML} terms that is entirely syntax directed: At each step in the derivation, we replace the type on the right-hand side of the ":" with its normal form. Given a procedure for determining normal forms, since the rest of the rules are syntax directed, type checking in the new proof system is entirely syntax directed. Furthermore, it is easy to see that the resulting proof system is equivalent to the original system. Any proof in the new system can be transformed into a proof in the old system simply by adding applications of the equivalence rule at each step, asserting that the type ascribed by the old system and its normal form are equivalent. Any proof in the old system can be transformed into a proof in the new system since normal forms are unique.

Consequently, if I can establish normal forms for constructors, show that two constructors are definitionally equivalent iff they have the same normal form, and give a procedure for finding normal forms, then we can use this procedure to formulate an entirely syntax-directed type checking system.

4.1.1 Reduction of Constructors

I begin by defining a reduction relation, $\mu \longrightarrow \mu'$, on "raw" constructors that (potentially) contain free variables. This primitive notion of reduction is generated from four rules: one rule each for β- and η-reduction of functionals, and two rules for reducing $\mathsf{Typerec}$s:

$$
\begin{array}{ll}
(\beta) & (\lambda t{::}\kappa.\,\mu)\;\mu' \longrightarrow \{\mu'/t\}\mu \\
(\eta) & (\lambda t{::}\kappa.\,(\mu\;t)) \longrightarrow \mu \qquad (t \notin FV(\mu)) \\
(\mathbf{t1}) & \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_{\mathrm{int}};\mu_{\mathrm{arrow}}) \longrightarrow \mu_{\mathrm{int}} \\
(\mathbf{t2}) & \mathsf{Typerec}\ \mathsf{Arrow}(\mu_1,\mu_2)\ \mathsf{of}\ (\mu_{\mathrm{int}};\mu_{\mathrm{arrow}}) \longrightarrow \\
 & \qquad \mu_{\mathrm{arrow}}\,\mu_1\,\mu_2\,(\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_{\mathrm{int}};\mu_{\mathrm{arrow}}))(\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_{\mathrm{int}};\mu_{\mathrm{arrow}}))
\end{array}
$$

I use $\mathbf{T}$ to abbreviate the union of these four relations:

$$\mathbf{T} = \beta \cup \eta \cup \mathbf{t1} \cup \mathbf{t2}$$

I extend the relation to form a congruence by defining term contexts which are arbitrary raw constructors with a single hole in them, much like constructor evaluation contexts, but with no restrictions governing where the hole can occur:

$$
\begin{array}{rcl}
C & ::= & [\,] \mid \mathsf{Arrow}(C,\mu) \mid \mathsf{Arrow}(\mu,C) \mid \lambda t{::}\kappa.\,C \mid C\,\mu \mid \mu\,C \mid \\
 & & \mathsf{Typerec}\ C\ \mathsf{of}\ (\mu_{\mathrm{int}};\mu_{\mathrm{arrow}}) \mid \mathsf{Typerec}\ \mu\ \mathsf{of}\ (C;\mu_{\mathrm{arrow}}) \mid \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_{\mathrm{int}};C)
\end{array}
$$

The constructor $C[\mu]$ represents the result of replacing the hole in C with the constructor μ, possibly binding free variables of μ.

Definition 4.1.1 *$\mu_1 \longrightarrow \mu_2$ iff there exists μ_1', μ_2', and C such that $\mu_1 = C[\mu_1']$, $\mu_2 = C[\mu_2']$, and $(\mu_1', \mu_2') \in \mathbf{T}$.*

I write $\mu_1 \longrightarrow^* \mu_2$ for the reflexive, transitive closure of $\mu_1 \longrightarrow \mu_2$, $\mu_1 \longleftrightarrow \mu_2$ for the symmetric closure of $\mu_1 \longrightarrow \mu_2$, and $\mu_1 \longleftrightarrow^* \mu_2$ for the least equivalence relation generated by $\mu_1 \longrightarrow^* \mu_2$. The constructor μ is in normal form if there is no μ' such that $\mu \longrightarrow \mu'$. I say that a constructor μ is strongly normalizing (SN) if there is no infinite reduction sequence $\mu \longrightarrow \mu_1 \longrightarrow \mu_2 \longrightarrow \cdots$.

Lemma 4.1.2

1. *If* $\mu_1 \longrightarrow \mu_2$, *then* $\{\mu_3/t\}\mu_1 \longrightarrow \{\mu_3/t\}\mu_2$.
2. *If* $\mu_1 \longrightarrow \mu_2$, *then* $\{\mu_1/t\}\mu_3 \longrightarrow \{\mu_2/t\}\mu_3$.

Proof (sketch): Straightforward induction on terms and case analysis of the reduction relation. □

The following lemmas relate the reduction relation to definitional equivalence. In particular, two well-formed constructors μ_1 and μ_2 are definitionally equivalent iff we can convert from μ_1 to μ_2 via the $\longleftrightarrow^*$ equivalence relation.

Lemma 4.1.3 (Substitution) *If* $\Delta \uplus \{t{::}\kappa'\} \vdash \mu :: \kappa$ *and* $\Delta \vdash \mu' :: \kappa'$, *then* $\Delta \vdash \{\mu'/t\}\mu :: \kappa$.

Lemma 4.1.4 *If* $\Delta \vdash C[\mu] :: \kappa$, *then there exists some* Δ' *and* κ' *such that* $\Delta' \vdash \mu :: \kappa'$ *and if* $\Delta' \vdash \mu' :: \kappa'$, *then* $\Delta \vdash C[\mu'] :: \kappa$.

Proof (sketch): By induction on C. □

Lemma 4.1.5 (Kind Preservation) *If* $\Delta \vdash \mu :: \kappa$ *and* $\mu \longrightarrow \mu'$, *then* $\Delta \vdash \mu' :: \kappa$.

Proof: Suppose $\Delta \vdash C[\mu] :: \kappa$. By the previous lemma, there exists some Δ' and κ' such that $\Delta' \vdash \mu :: \kappa'$. Hence, it suffices to show that $(\mu, \mu') \in \mathbf{T}$ implies $\Delta' \vdash \mu' :: \kappa'$.

β : $\mu = (\lambda t{::}\kappa_1.\, \mu_1)\, \mu_2$ and $\mu' = \{\mu_2/t\}\mu_1$. Follows from the typing rule for functions and the Substitution Lemma.

η : $\mu = \lambda t{::}\kappa_1.\, (\mu_1\, t)$ $(t \notin FV(\mu_1))$ and $\mu' = \mu_1$. By the typing rule for functions, $\Delta' \uplus \{t{::}\kappa_1\} \vdash \mu_1\, t :: \kappa_2$ for some κ_2 and $\kappa' = \kappa_1 \to \kappa_2$. By the application rule, $\Delta' \uplus \{t{::}\kappa_1\} \vdash \mu_1 :: \kappa'$. Since t does not occur free in μ_1, $\Delta' \vdash \mu_1 :: \kappa'$.

t1 : $\mu = \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_i; \mu_a)$ and $\mu' = \mu_i$. By the typing rule for Typerec, $\Delta' \vdash \mu :: \kappa'$ and $\Delta \vdash \mu_i :: \kappa'$.

t2 : $\mu = \mathsf{Typerec}\ \mathsf{Arrow}(\mu_1, \mu_2)\ \mathsf{of}\ (\mu_i; \mu_a)$ and

$$\mu' = \mu_a\, \mu_1\, \mu_2\, \mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i; \mu_a)\ \mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i; \mu_a).$$

By the typing rule for Typerec, $\Delta' \vdash \mu :: \kappa'$ and $\Delta \vdash \mu_i :: \kappa'$, $\Delta' \vdash \mu_a :: \Omega \to \Omega \to \kappa' \to \kappa' \to \kappa'$. By the rule for Arrow, $\Delta' \vdash \mu_1 :: \Omega$ and $\Delta' \vdash \mu_2 :: \Omega$. By the application rule, $\mu_a\, \mu_1\, \mu_2$ has type $\kappa' \to \kappa' \to \kappa'$. By the typing rule for Typerec, $\Delta' \vdash \mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i; \mu_a) :: \kappa'$ and likewise for μ_2. Thus, by the application rule, $\Delta' \vdash \mu' :: \kappa'$.

□

Lemma 4.1.6 *If* $\Delta \vdash \mu :: \kappa$ *and* $\mu \longrightarrow \mu'$*, then* $\Delta \vdash \mu \equiv \mu' :: \kappa$.

Proof (sketch): Suppose $\Delta \vdash C[\mu] :: \kappa$ and $(\mu, \mu') \in \mathbf{T}$. Then there exists some Δ', κ' such that $\Delta' \vdash \mu :: \kappa'$ and by preservation, $\Delta' \vdash \mu' :: \kappa'$. Argue by cases that $\Delta' \vdash \mu \equiv \mu' :: \kappa'$. Then show by induction on C that $\Delta' \vdash \mu \equiv \mu' :: \kappa'$ implies $\Delta \vdash C[\mu] \equiv C[\mu'] :: \kappa$. □

Lemma 4.1.7 *If* $\Delta \vdash \mu_1 :: \kappa$*,* $\Delta \vdash \mu_2 :: \kappa$*, and there exists a* μ *such that* $\mu_1 \longrightarrow^* \mu$ *and* $\mu_2 \longrightarrow^* \mu$*, then* $\Delta \vdash \mu_1 \equiv \mu_2 : \kappa$.

Proof: By the previous lemma, $\Delta \vdash \mu_1 \equiv \mu :: \kappa$ and $\Delta \vdash \mu_2 \equiv \mu :: \kappa$ and by symmetry and transitivity, $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$. □

Lemma 4.1.8 $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$ *iff* $\mu_1 \longleftrightarrow^* \mu_2$.

Proof: The "only-if" is apparent from the previous lemma. Hence, I must show that $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$ implies $\mu_1 \longleftrightarrow^* \mu_2$. I argue by induction on the derivation of $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$. Reflexivity, symmetry, and transitivity follow from the definition of $\longleftrightarrow^*$. The arrow, fn, app, and trec rules follow by building an appropriate context C for each of the component constructors μ, using the induction hypothesis to argue that $\mu \equiv \mu'$ implies $\mu \longleftrightarrow^* \mu'$ and thus $C[\mu] \equiv C[\mu']$. The subcases are glued together via transitivity. The β, η, trec-int, and trec-arrow rules all follow from their counterparts in $\mathbf{T}$. □

The main results of this chapter are that the reduction relation "$\longrightarrow^*$" is *confluent* and strongly normalizing for well-formed constructors. That is, every reduction sequence terminates and the reduction sequences have the diamond property: If a constructor reduces to two different constructors, then those two constructors reduce to a common constructor.

Proposition 4.1.9 (Strong Normalization) *If* $\Delta \vdash \mu :: \kappa$*, then* μ *is strongly normalizing.*

Proposition 4.1.10 (Confluence) *If* $\mu \longrightarrow^* \mu_1$ *and* $\mu \longrightarrow^* \mu_2$*, then there exists a constructor* μ' *such that* $\mu_1 \longrightarrow^* \mu'$ *and* $\mu_2 \longrightarrow^* \mu'$*.*

Confluence for "$\longrightarrow^*$" is important because of the following immediate corollaries: First, normal forms are unique up to α-conversion:

Corollary 4.1.11 *If* $\mu \longrightarrow^* \mu_1$ *and* $\mu \longrightarrow^* \mu_2$ *and* μ_1 *and* μ_2 *are normal forms, then* $\mu_1 = \mu_2$*.*

Second, the reduction system has the "Church-Rosser" property, which tells us that finding and comparing normal forms is a *complete* procedure for determining equivalence of constructors.

Corollary 4.1.12 (Church-Rosser) $\mu \longleftrightarrow^* \mu'$ *iff there exists a* μ'' *such that* $\mu \longrightarrow^* \mu''$ *and* $\mu' \longrightarrow^* \mu''$*.*

Proof: The "if" part is obvious. For the "only if", I argue by induction on the length of the sequence $\mu \longleftrightarrow \mu_1 \longleftrightarrow \cdots \longleftrightarrow \mu_n \longleftrightarrow \mu'$. For $n = 0$, $\mu \longleftrightarrow \mu'$ and thus either $\mu \longrightarrow \mu'$ or else $\mu' \longrightarrow \mu$. Without loss of generality, assume $\mu \longrightarrow \mu'$. Then choose $\mu'' = \mu'$ and we are done. Assume the theorem holds for all values up through n. We have $\mu \longleftrightarrow \mu_1 \longleftrightarrow \cdots \longleftrightarrow \mu_n$ and $\mu_n \longleftrightarrow \mu'$. I must show that there is a μ'' such that $\mu_n \longrightarrow^* \mu''$ and $\mu' \longrightarrow^* \mu''$. By the induction hypothesis, there exists μ_a and μ_b such that $\mu \longrightarrow^* \mu_a$, $\mu_n \longrightarrow^* \mu_a$, $\mu_n \longrightarrow^* \mu_b$, and $\mu' \longrightarrow^* \mu_b$. Since μ_n reduces to both μ_a and μ_b, we have via confluence that there exists a μ'' such that $\mu_a \longrightarrow^* \mu''$ and $\mu_b \longrightarrow^* \mu''$ and hence $\mu \longrightarrow^* \mu''$ and $\mu' \longrightarrow^* \mu''$. □

Since for well-formed constructors, convertibility is the same as definitional equivalence, the Church-Rosser property implies that two well-formed constructors are definitionally equivalent iff there is some common reduct. If I can show that all well-formed constructors are strongly normalizing, then we have a decision procedure for determining constructor equivalence: Choose any reduction sequence for the two constructors and eventually, we will reach normal forms, since the two constructors are strongly normalizing. Then, the Church-Rosser theorem together with unique normal forms, tells us that the two constructors are equivalent iff the two normal forms are equivalent modulo α-conversion.

In the presence of strong normalization, confluence is equivalent to *local confluence*: If $\mu \longrightarrow \mu_1$ and $\mu \longrightarrow \mu_2$, then there exists a μ' such that $\mu_1 \longrightarrow^* \mu'$ and $\mu_2 \longrightarrow^* \mu'$.

Lemma 4.1.13 *If* μ *is strongly normalizing and locally confluent, then* μ *is confluent.*

Proof: Suppose $\mu \longrightarrow^* \mu_1$ in m steps and $\mu \longrightarrow^* \mu_2$ in n steps. Since μ is strongly normalizing, there exists some bound b on the number of steps any reduction sequence can take. I argue by induction on b and reduce the problem to $m \leq 1$, $n \leq 1$. If $m = 0$ or $n = 0$ then there is nothing to prove. The case $m = 1$, $n = 1$ is handled by local confluence.

Suppose $m > 1$ or else $n > 1$. Then, we have $\mu \longrightarrow \mu'_1 \longrightarrow^{m-1} \mu_1$ and $\mu \longrightarrow \mu'_2 \longrightarrow^{n-1} \mu_2$ where $m - 1 > 0$ or $n - 1 > 0$. By local confluence, we know there is a μ'' such that $\mu'_1 \longrightarrow^* \mu''$ and $\mu'_2 \longrightarrow^* \mu''$.

Now μ'_1 and μ'_2 have smaller bounds than μ, so by the induction hypothesis, there exists μ''_1 and μ''_2 such that $\mu_1 \longrightarrow^* \mu''_1$, $\mu'' \longrightarrow^* \mu''_1$, $\mu_2 \longrightarrow^* \mu''_2$, and $\mu'' \longrightarrow^* \mu''_2$.

Again, μ'' has a bound less than b, so applying the induction hypothesis, we know that there exists a μ' such that $\mu''_1 \longrightarrow^* \mu'$ and $\mu''_2 \longrightarrow^* \mu'$. Thus, $\mu_1 \longrightarrow^* \mu'$ and $\mu_2 \longrightarrow^* \mu'$.

The result is summarized by the following diagram, where assuming the solid arrows, we have shown that the dotted arrows exist:

$$
\begin{tikzcd}
 & & \mu \arrow[dl, "1"'] \arrow[dr, "1"] & & \\
 & \mu'_1 \arrow[dl, "m-1"'] \arrow[dr, dotted, "*"] & & \mu'_2 \arrow[dl, dotted, "*"'] \arrow[dr, "n-1"] & \\
\mu_1 \arrow[dr, dotted, "*"] & & \mu'' \arrow[dl, dotted, "*"'] \arrow[dr, dotted, "*"] & & \mu_2 \arrow[dl, dotted, "*"'] \\
 & \mu''_1 \arrow[dr, dotted, "*"] & & \mu''_2 \arrow[dl, dotted, "*"'] & \\
 & & \mu' & &
\end{tikzcd}
$$

□

All that remains is to establish local confluence and strong normalization, which I address in Sections 4.1.2 and 4.1.3, respectively.

4.1.2 Local Confluence for Constructor Reduction

To show that the reduction system for constructors is locally confluent, I must show that whenever $\mu \longrightarrow \mu_1$ and $\mu \longrightarrow \mu_2$, then there exists a μ' such that $\mu_1 \longrightarrow^* \mu'$ and $\mu_2 \longrightarrow^* \mu'$.

Let D range over the set of expressions with exactly two holes in them. This set can be described by the following grammar:

$$\begin{array}{rcl} D & ::= & \mathsf{Arrow}(C_1, C_2) \mid C_1 C_2 \mid \mathsf{Typerec}\ C_1\ \mathsf{of}\ (C_2; \mu_{\text{arrow}}) \mid \\ & & \mathsf{Typerec}\ C_1\ \mathsf{of}\ (\mu_{\text{int}}; C_2) \mid \mathsf{Typerec}\ \mu\ \mathsf{of}\ (C_1, C_2) \mid C[D] \end{array}$$

where I use $C[D]$ to denote the two-holed constructor formed by replacing the hole in C with D and I use $D[\mu_1, \mu_2]$ to denote the constructor obtained by replacing the left-most hole in D with μ_1 and the right-most hole in D with μ_2.

Suppose $\mu = C_a[\mu_a]$, $(\mu_a, \mu_a') \in \mathbf{T}$, and $\mu = C_b[\mu_b]$ and $(\mu_b, \mu_b') \in \mathbf{T}$. The two constructors, μ_a and μ_b are said to *overlap* in μ if one constructor is only found as a subterm of the other (i.e., $\mu_a = C[\mu_b]$ or $\mu_b = C[\mu_a]$ for some C). If μ_a and μ_b do not overlap, then it is clear that there exists some D such that $\mu = D[\mu_a, \mu_b]$ and thus:

$$\mu = D[\mu_a, \mu_b] \longrightarrow D[\mu_a', \mu_b] \longrightarrow D[\mu_a', \mu_b']$$

and

$$\mu = D[\mu_a, \mu_b] \longrightarrow D[\mu_a, \mu_b'] \longrightarrow D[\mu_a', \mu_b'].$$

Therefore, we need only consider overlapping constructors to show local confluence.

Let μ_a and μ_b be overlapping constructors in μ, such that $\mu = C_a[\mu_a]$, $\mu_a = C_b[\mu_b]$, and suppose $(\mu_a, \mu_a') \in \mathbf{T}$ and $(\mu_b, \mu_b') \in \mathbf{T}$. Without loss of generality, we may ignore the outer context, C_a. There are four cases for the reduction from μ_a to μ_a'. I consider each case below and show that, for each rule taking μ_b to μ_b', there exists a μ' and sequence of reductions which takes μ_a' to μ' and $C_b[\mu_b']$ to μ'. Each argument is made by presenting a diagram where the left-arrow represents the reduction from μ_a to μ_a', and the right arrow represents the reduction from $\mu_a = C_b[\mu_b]$ to $C_b[\mu_b']$. The solid arrows represent assumptions, whereas the dotted arrows represent the relations I claim exist. I use $\mathbf{t}$ to represent some arbitrary reduction from $\mathbf{T}$, α to represent α conversion, $=$ to represent zero reductions, and $\mathbf{t}^*$ to represent zero or more applications of a $\mathbf{t}$ reduction.

case β: Both sub-cases follow from lemma 4.1.2, part 1 and 2 respectively.

$$\begin{tikzcd}
& (\lambda t{::}\kappa.\, \mu_1)\, \mu_2 \arrow[dl, "\beta"'] \arrow[dr, "\mathbf{t}"] & \\
\{\mu_2/t\}\mu_1 \arrow[dr, dotted, "\mathbf{t}"'] & & (\lambda t{::}\kappa.\, \mu_1')\, \mu_2 \arrow[dl, dotted, "\beta"] \\
& \{\mu_2/t\}\mu_1' &
\end{tikzcd}
\qquad
\begin{tikzcd}
& (\lambda t{::}\kappa.\, \mu_1)\, \mu_2 \arrow[dl, "\beta"'] \arrow[dr, "\mathbf{t}"] & \\
\{\mu_2/t\}\mu_1 \arrow[dr, dotted, "\mathbf{t}^*"'] & & (\lambda t{::}\kappa.\, \mu_1)\, \mu_2' \arrow[dl, dotted, "\beta"] \\
& \{\mu_2'/t\}\mu_1 &
\end{tikzcd}$$

case η: The constructor is an η-redex $\lambda t{::}\kappa.\,(\mu_1\,t)$. If the inner reduction occurs within μ_1, then the result is obvious, since t cannot occur freely within μ_1. If the inner reduction is an application of β because μ_1 is a function, then the results of the outer reduction and inner reduction yield terms that are equal, modulo α-conversion.

$$\begin{tikzcd} & \lambda t{::}\kappa.\,(\mu_1\,t) \arrow[dl, "\eta"'] \arrow[dr, "\mathbf{t}"] & \\ \mu_1 \arrow[dr, dotted, "\mathbf{t}"'] & & \lambda t{::}\kappa.\,(\mu_1'\,t) \arrow[dl, dotted, "\eta"] \\ & \mu_1' & \end{tikzcd} \qquad \begin{tikzcd} & \lambda t{::}\kappa.\,((\lambda t'{::}\kappa'.\,\mu_1)\,t) \arrow[dl, "\eta"'] \arrow[dr, "\beta"] & \\ \lambda t'{::}\kappa'.\,\mu_1 \arrow[dr, dotted, "="'] & & \lambda t{::}\kappa.\,\{t/t'\}\mu_1 \arrow[dl, dotted, "\alpha"] \\ & \lambda t'{::}\kappa'.\,\mu_1 & \end{tikzcd}$$

case t1: The constructor is a $\mathsf{Typerec}$ applied to Int. The inner reduction either modifies μ_i or μ_a. If μ_i is reduced, then after performing the $\mathsf{Typerec}$ reduction, the same reduction can be applied. If μ_a is reduced, then after performing the $\mathsf{Typerec}$ reduction, μ_a disappears, and the terms are equal.

$$\begin{tikzcd} & \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, "\mathbf{t1}"'] \arrow[dr, "\mathbf{t}"] & \\ \mu_i \arrow[dr, dotted, "\mathbf{t}"'] & & \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_i';\mu_a) \arrow[dl, dotted, "\mathbf{t1}"] \\ & \mu_i' & \end{tikzcd}$$

$$\begin{tikzcd} & \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, "\mathbf{t1}"'] \arrow[dr, "\mathbf{t}"] & \\ \mu_i \arrow[dr, dotted, "="'] & & \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_i;\mu_a') \arrow[dl, dotted, "\mathbf{t1}"] \\ & \mu_i & \end{tikzcd}$$

case t2: The constructor is a $\mathsf{Typerec}$ applied to $\mathsf{Arrow}(\mu_1,\mu_2)$. There are four subcases: μ_1, μ_2, μ_i, or μ_a is reduced. In any of these cases, we can apply the same reduction, possibly multiple times to yield the same term.

$$
\begin{tikzcd}[column sep=tiny]
& \mathsf{Typerec\ Arrow}(\mu_1,\mu_2)\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, "\mathbf{t2}"'] \arrow[dr, "\mathbf{t}"] & \\
\begin{array}{c}\mu_a\,\mu_1\,\mu_2\,(\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i;\mu_a))\\ (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i;\mu_a))\end{array} \arrow[dr, dotted, "\mathbf{t}^*"'] & & \mathsf{Typerec\ Arrow}(\mu_1',\mu_2)\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, dotted, "\mathbf{t2}"] \\
& \mu_a\,\mu_1'\,\mu_2\,(\mathsf{Typerec}\ \mu_1'\ \mathsf{of}\ (\mu_i;\mu_a))\,(\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i;\mu_a)) &
\end{tikzcd}
$$

$$
\begin{tikzcd}[column sep=tiny]
& \mathsf{Typerec\ Arrow}(\mu_1,\mu_2)\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, "\mathbf{t2}"'] \arrow[dr, "\mathbf{t}"] & \\
\begin{array}{c}\mu_a\,\mu_1\,\mu_2\,(\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i;\mu_a))\\ (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i;\mu_a))\end{array} \arrow[dr, dotted, "\mathbf{t}^*"'] & & \mathsf{Typerec\ Arrow}(\mu_1,\mu_2')\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, dotted, "\mathbf{t2}"] \\
& \mu_a\,\mu_1\,\mu_2'\,(\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i;\mu_a))\,(\mathsf{Typerec}\ \mu_2'\ \mathsf{of}\ (\mu_i;\mu_a)) &
\end{tikzcd}
$$

$$
\begin{tikzcd}[column sep=tiny]
& \mathsf{Typerec\ Arrow}(\mu_1,\mu_2)\ \mathsf{of}\ (\mu_i;\mu_a) \arrow[dl, "\mathbf{t2}"'] \arrow[dr, "\mathbf{t}"] & \\
\begin{array}{c}\mu_a\,\mu_1\,\mu_2\,(\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i;\mu_a))\\ (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i;\mu_a))\end{array} \arrow[dr, dotted, "\mathbf{t}^*"'] & & \mathsf{Typerec\ Arrow}(\mu_1,\mu_2)\ \mathsf{of}\ (\mu_i';\mu_a) \arrow[dl, dotted, "\mathbf{t2}"] \\
& \mu_a\,\mu_1\,\mu_2\,(\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i';\mu_a))\,(\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i';\mu_a)) &
\end{tikzcd}
$$

$$
\begin{tikzcd}[column sep=small]
 & \textsf{Typerec Arrow}(\mu_1,\mu_2)\ \textsf{of}\ (\mu_i;\mu_a) \arrow[dl, "\mathbf{t2}"'] \arrow[dr, "\mathbf{t}"] & \\
\begin{array}{c}\mu_a\,\mu_1\,\mu_2\,(\textsf{Typerec}\ \mu_1\ \textsf{of}\ (\mu_i;\mu_a)) \\ (\textsf{Typerec}\ \mu_2\ \textsf{of}\ (\mu_i;\mu_a))\end{array} \arrow[dr, dotted, "\mathbf{t}^*"'] & & \textsf{Typerec Arrow}(\mu_1,\mu_2)\ \textsf{of}\ (\mu_i;\mu_a') \arrow[dl, dotted, "\mathbf{t2}"] \\
 & \mu_a'\,\mu_1\,\mu_2\,(\textsf{Typerec}\ \mu_1\ \textsf{of}\ (\mu_i;\mu_a'))\,(\textsf{Typerec}\ \mu_2\ \textsf{of}\ (\mu_i;\mu_a')) &
\end{tikzcd}
$$

Finally, from the diagrams above, we have the desired property.

Theorem 4.1.14 (Local Confluence) *If $\mu \longrightarrow \mu_1$ and $\mu \longrightarrow \mu_2$, then there exists a μ' such that $\mu_1 \longrightarrow^* \mu'$ and $\mu_2 \longrightarrow^* \mu_2$.*

All that remains is to show that well-formed constructors are strongly normalizing.

4.1.3 Strong Normalization for Constructor Reduction

Our proof of strong normalization for the constructor reduction relation uses unary logical relations (predicates), but in a setting where we can have open terms. The ideas follow closely those of Harper [55] and Lambek and Scott [79].

The predicates are indexed by both a kind (κ) and a kind assignment (Δ) and are defined as follows:

$$
\begin{array}{rcl}
||\Omega||_\Delta & = & \{\mu \mid \Delta \vdash \mu :: \kappa, \mu\ SN\} \\
||\kappa_1 \to \kappa_2||_\Delta & = & \{\mu \mid \Delta \vdash \mu :: \kappa_1 \to \kappa_2, \forall \Delta' \supseteq \Delta.\, \forall \mu_1 \in ||\kappa_1||_{\Delta'}.\, \mu\,\mu_1 \in ||\kappa_2||_{\Delta'}\}
\end{array}
$$

The idea is to include only those constructors that are strongly normalizing and then show that every well-formed constructor is in the appropriate set. From the definitions, it is clear that if $\mu \in ||\kappa||_\Delta$ then $\Delta \vdash \mu :: \kappa$ and for all $\Delta' \supseteq \Delta$, $||\kappa||_{\Delta'}$.

Lemma 4.1.15 *$t\,\mu_1\,\mu_2 \cdots \mu_n$ is SN iff each μ_i is SN.*

The following lemma shows that every constructor in one of the sets is strongly normalizing and that a variable t is always in $||\kappa||_\Delta$, whenever $\Delta(t) = \kappa$.

Lemma 4.1.16 1. *If $\mu \in ||\kappa||_\Delta$, then μ is SN.*

2. *If* $\Delta \vdash t :: \kappa_1 \rightarrow \cdots \rightarrow \kappa_n \rightarrow \kappa$, $\Delta' \vdash \mu_i :: \kappa_i$ *and* μ_i *is* SN $(1 \leq i \leq n)$ *for some* $\Delta' \supseteq \Delta$, *then* $t\,\mu_1 \cdots \mu_n \in ||\kappa||_{\Delta'}$.

Proof: Simultaneously by induction on κ. If $\kappa = \Omega$, then part 1 is built in to the definition and part 2 follows since $t\,\mu_1 \cdots \mu_n$ is SN whenever each of the μ_i are SN.

Suppose $\kappa = \kappa' \rightarrow \kappa''$ and $\mu \in ||\kappa||_\Delta$. Take $\Delta' = \Delta \uplus \{t{::}\kappa'\}$. By induction hypothesis 2, $t \in ||\kappa'||_{\Delta'}$ and $\mu\, t \in ||\kappa''||_{\Delta'}$ from which it follows via induction hypothesis 1 that $\mu\, t$ is SN. This in turn implies that μ is SN.

Suppose $\kappa = \kappa' \rightarrow \kappa''$, $\Delta \vdash t :: \kappa_1 \rightarrow \cdots \rightarrow \kappa_n \rightarrow \kappa$, and $\Delta' \vdash \mu_i :: \kappa_i$ is SN for some $\Delta' \supseteq \Delta$. Let $\mu \in ||\kappa'||_{\Delta'}$. I must show that $t\,\mu_1 \cdots \mu_n\, \mu \in ||\kappa''||_{\Delta'}$. But by induction hypothesis 1, μ is SN. Hence, by induction hypothesis 2, the result holds. $\Box$

Corollary 4.1.17 *If* $\Delta \vdash t :: \kappa$, *then* $t \in ||\kappa||_\Delta$.

The following lemma shows that the predicates are closed under a suitable notion of β-expansion. This lemma is crucial for showing that well-formed λ-terms are in the sets and therefore are strongly normalizing.

Lemma 4.1.18 *Suppose* $\Delta \uplus \{t{::}\kappa_0\} \vdash \mu :: \kappa_1 \rightarrow \cdots \rightarrow \kappa_n \rightarrow \kappa$ *and* $\mu_i \in ||\kappa_i||_\Delta$ $(0 \leq i \leq n)$. *If* $(\{\mu_0/t\}\mu)\,\mu_1 \cdots \mu_n \in ||\kappa||_\Delta$, *then* $(\lambda t{::}\kappa_0.\,\mu)\,\mu_0\,\mu_1 \cdots \mu_n \in ||\kappa||_\Delta$.

Proof:

By induction on κ. Assume $\kappa = \Omega$. I must show $\mu_a = (\lambda t{::}\kappa_0.\,\mu)\,\mu_0\,\mu_1 \cdots \mu_n$ is in $||\Omega||_\Delta$. It suffices to show that this term is SN. Suppose not. Then there exists some infinite reduction sequence starting with μ_a. Since $(\{\mu_0/t\}\mu)\,\mu_1 \cdots \mu_n \in ||\Omega||_\Delta$, this term is SN and hence all of its components are. Moreover, since $\mu_0 \in ||\kappa_0||_\Delta$, μ_0 is SN. Hence, no infinite reduction sequence can take place only within $\mu, \mu_0, \mu_1, \cdots, \mu_n$. Thus, any infinite reduction sequence has the form:

$$\begin{array}{l}(\lambda t{::}\kappa_0.\,\mu)\,\mu_0\,\mu_1 \cdots \mu_n \longrightarrow^* \\ \qquad (\lambda t{::}\kappa_0.\,\mu')\,\mu_0'\,\mu_1' \cdots \mu_n' \longrightarrow \\ \qquad\qquad (\{\mu_0'/t\}\mu'), \mu_1', \cdots, \mu_n' \longrightarrow \mu'' \longrightarrow \cdots\end{array}$$

Consequently, $\mu \longrightarrow^* \mu'$, $\mu_0 \longrightarrow^* \mu_0'$, and $\{\mu_0/t\}\mu \longrightarrow^* \{\mu_0'/t\}\mu'$. Thus, we can construct an infinite reduction sequence:

$$\begin{array}{l}(\{\mu_0/t\}\mu\,\mu_1 \cdots \mu_n \longrightarrow^* \\ \qquad (\{\mu_0'/t\}\mu'\,\mu_1' \cdots \mu_n' \longrightarrow \mu'' \longrightarrow \cdots\end{array}$$

contradicting the assumption. Therefore, $\mu_a \in ||\Omega||_\Delta$.

Assume $\kappa = \kappa' \to \kappa''$, and let $\mu' \in ||\kappa'||_{\Delta'}$ for $\Delta' \supseteq \Delta$. I must show that

$$(\lambda t{::}\kappa_0.\, \mu)\, \mu_0\, \mu_1\, \cdots\, \mu_n\, \mu'$$

is in $||\kappa''||_{\Delta'}$. By the induction hypothesis, this holds if

$$(\{\mu_0/t\}\mu)\, \mu_1\, \cdots\, \mu_n\, \mu'$$

is in $||\kappa''||_{\Delta'}$. But this follows from the assumption that

$$(\{\mu_0/t\}\mu)\, \mu_1\, \cdots\, \mu_n$$

is in $||\kappa' \to \kappa''||_{\Delta}$. $\square$

Corollary 4.1.19 *If* $\Delta \uplus \{t{::}\kappa_0\} \vdash \mu :: \kappa$, $\mu_0 \in ||\kappa_0||_\Delta$, *and* $\{\mu_0/t\}\mu \in ||\kappa||_\Delta$, *then* $(\lambda t{::}\kappa_0.\, \mu)\, \mu_0 \in ||\kappa||_\Delta$.

The following lemma shows that the predicates are closed under a suitable notion of Typerec-expansion. This lemma is crucial for showing that well-formed Typerec-constructors are in the sets and hence are strongly normalizing.

Lemma 4.1.20 *Let* $\kappa' = \kappa_1 \to \cdots \to \kappa_n \to \kappa$ *and suppose* $\Delta \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a) :: \kappa'$, $\mu_j \in ||\kappa_j||_\Delta$ *(*$1 \leq j \leq n$*)*, $\mu \in ||\Omega||_\Delta$, $\mu_i \in ||\kappa'||_\Delta$, *and* $\mu_a \in ||\Omega \to \Omega \to \kappa' \to \kappa' \to \kappa'||_\Delta$. *Then* $\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a) \in ||\kappa'||_\Delta$.

Proof:
By induction on κ. If $\kappa = \Omega$, then it suffices to show that

$$\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a)\, \mu_1\, \cdots\, \mu_n$$

is SN. I argue by induction on the height (h) of the normal form of μ. Suppose $h = 0$. Then the normal form of μ is either a variable or Int. Since $\mu \in ||\Omega||_\Delta$, $\mu_i \in ||\kappa'||_\Delta$, μ_a is in

$$||\Omega \to \Omega \to \kappa' \to \kappa' \to \kappa'||_\Delta,$$

and $\mu_j \in ||\kappa_j||_\Delta$ $(1 \leq j \leq n)$, μ, μ_i, μ_a, and $\mu_1, \cdots, \mu_n$ are all SN. Hence, any infinite reduction sequence cannot occur only within these terms. Thus, any such sequence must perform a Typerec reduction and the normal form of μ must be Int, so the infinite reduction sequence has the form:

$$\begin{array}{l} \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a)\, \mu_1\, \cdots\, \mu_n \longrightarrow^* \\ \qquad \mathsf{Typerec}\ \mathsf{Int}\ \mathsf{of}\ (\mu_i'; \mu_a')\, \mu_1' \cdots \mu_n' \longrightarrow \mu_i'\, \mu_1'\, \cdots\, \mu_n' \longrightarrow \cdots \end{array}$$

But since $\mu_i \in ||\kappa'||_\Delta$, and $\mu_j \in ||\kappa_j||_\Delta$ $(1 \leq j \leq n)$, $\mu_i' \, \mu_1' \, \cdots \, \mu_n'$ is SN. Therefore, $\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a) \in ||\kappa'||_\Delta$ when $h = 0$.

Suppose the theorem holds for all terms with normal forms of height less than h and suppose the normal form of μ has height $h + 1$. By the same previous argument, any infinite reduction sequence must perform a Typerec reduction. Furthermore, since the normal form of μ has height $h + 1$, such a sequence must have the form:

$$\begin{array}{l} \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a)\, \mu_1 \cdots \mu_n \longrightarrow^* \\ \quad \mathsf{Typerec}\ \mathsf{Arrow}(\mu_1, \mu_2)\ \mathsf{of}\ (\mu_i'; \mu_a')\, \mu_1' \cdots \mu_n' \longrightarrow \\ \quad ((\mu_a'\, \mu_1\, \mu_2\, (\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i'; \mu_a'))\, (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i'; \mu_a')))\, \mu_1' \cdots \mu_n') \longrightarrow \cdots \end{array}$$

Since μ is SN, it must be the case that μ_1 and μ_2 are SN, and hence $\mu_1, \mu_2 \in ||\Omega||_\Delta$. Furthermore, the heights of the normal forms of μ_1 and μ_2 must be less than or equal to h. Hence, via the inner induction hypothesis, $\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i'; \mu_a') \in ||\Omega||_\Delta$ and $\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i'; \mu_a') \in ||\Omega||_\Delta$. Since by assumption $\mu_a \in ||\Omega \to \Omega \to \kappa' \to \kappa' \to \kappa'||_\Delta$,

$$(\mu_a'\, \mu_1\, \mu_2\, (\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i'; \mu_a'))\, (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i'; \mu_a')))\, \mu_1' \cdots \mu_n'$$

is SN and in $||\Omega||_\Delta$.

Now suppose $\kappa = \kappa' \to \kappa''$ and let $\mu' \in ||\kappa'||_{\Delta'}$ for some $\Delta \supseteq \Delta$. I must show that $(\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a))\, \mu_1 \cdots \mu_n\, \mu' \in ||\kappa''||_{\Delta'}$. But this follows from the outer induction hypothesis. □

Corollary 4.1.21 *If $\Delta \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a) :: \kappa$, $\mu \in ||\Omega||_\Delta$, $\mu_i \in ||\kappa||_\Delta$, and $\mu_a \in ||\Omega \to \Omega \to \kappa \to \kappa \to \kappa||_\Delta$, then $\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a) \in ||\kappa||_\Delta$.*

Let δ range over substitutions of constructors for constructor variables.

Definition 4.1.22

1. *$\Delta' |\!\vdash \mu :: \kappa[\delta]$ iff $\delta(\mu) \in ||\kappa||_{\Delta'}$.*
2. *$\Delta' |\!\vdash \delta :: \Delta$ iff $Dom(\delta) = Dom(\Delta)$ and for each t in $Dom(\delta)$, $\delta(t) \in ||\Delta(t)||_{\Delta'}$.*
3. *$|\!\vdash \Delta \vdash \mu :: \kappa$ iff for every Δ' and every δ such that $\Delta' |\!\vdash \delta :: \Delta$, $\Delta' |\!\vdash \mu :: \kappa[\delta]$.*

Theorem 4.1.23 *If $\Delta \vdash \mu :: \kappa$, then $|\!\vdash \Delta \vdash \mu :: \kappa$.*

Proof: By induction on the derivation of $\Delta \vdash \mu :: \kappa$. Suppose $\Delta' |\!\vdash \delta :: \Delta$.
var: Holds by the assumption $\Delta' |\!\vdash \delta$.
int: Holds, since $\delta(\mathsf{Int}) = \mathsf{Int}$ is trivially SN and thus $\mathsf{Int} \in ||\Omega||_{\Delta'}$.

arrow: Must show $\Delta' | \vdash \mathsf{Arrow}(\mu_1, \mu_2) :: \Omega[\delta]$. By the induction hypothesis, $\delta(\mu_1) \in ||\Omega||_{\Delta'}$ and $\delta(\mu_2) \in ||\Omega||_{\Delta'}$. Hence, μ_1 and μ_2 are SN and $\mathsf{Arrow}(\mu_1, \mu_2)$ is SN. Thus, $\mathsf{Arrow}(\mu_1, \mu_2) \in ||\Omega||_{\Delta'}$.
fn: Must show $\delta(\lambda t{::}\kappa_1.\, \mu) \in ||\kappa_1 \to \kappa_2||_{\Delta'}$. Let $\mu_1 \in ||\kappa_1||_{\Delta''}$ for some $\Delta'' = \Delta' \uplus \{t{::}\kappa_1\}$. I must show that $\{\mu_1/t\}\delta(\mu) \in ||\kappa_2||_{\Delta''}$. By the induction hypothesis, $\delta\mu \in ||\kappa_2||_{\Delta''}$ so the result follows from lemma 4.1.19
app: Must show $\delta(\mu_1\, \mu_2) \in ||\kappa_2||_{\Delta'}$. By the induction hypotheses, $\delta(\mu_1) \in ||\kappa_1 \to \kappa_2||_{\Delta'}$ and $\delta(\mu_2) \in ||\kappa_1||_{\Delta'}$, so the result follows from the definition of $||\kappa_1 \to \kappa_2||_{\Delta'}$.
trec: Must show $\delta(\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a)) \in ||\kappa||_{\Delta'}$. By the induction hypotheses, $\delta(\mu) \in ||\Omega||_{\Delta'}$, $\delta(\mu_i) \in ||\kappa||_{\Delta'}$, and $\delta(\mu_a) \in ||\Omega \to \Omega \to \kappa \to \kappa \to \kappa||_{\Delta'}$, so the result follows from lemma 4.1.21. □

Corollary 4.1.24 (Strong Normalization) *If $\Delta \vdash \mu :: \kappa$, then μ is strongly normalizing.*

Proof: Pick δ to be the identity substitution for Δ. That is, $\delta = \{t{=}t \mid t \in Dom(\Delta)\}$. Then it is easy to see that $\Delta | \vdash \delta :: \Delta$. Moreover, $\delta(\mu) = \mu \in ||\kappa||_{\Delta}$ and thus μ is SN. □

Corollary 4.1.25

1. *Every constructor μ has a unique normal form, $NF(\mu)$.*
2. *If μ is well-formed, there is an algorithm to calculate $NF(\mu)$.*
3. *Conversion of well-formed constructors is decidable.*

4.1.4 Decidability of Type Checking

In the following definitions, I establish a suitable notion of a normalized derivation for typing judgments. I then show that a term is well typed iff there is a normal derivation of the judgment. The proof is constructive, and thus provides an algorithm for type checking λ_i^{ML}.

Definition 4.1.26 (Normal Types, Judgments) *A type σ is in normal form iff it is* int, *$T(\mu)$ where μ is normal, $\sigma_1 \to \sigma_2$ where σ_1 and σ_2 are normal, or $\forall t{::}\kappa.\sigma'$ where σ' is normal. A judgment $\Delta; \Gamma \vdash e : \sigma$ is normal iff σ is normal.*

From the properties of constructors, it is clear that every well-formed type σ, has a unique normal form $NF(\sigma)$, and that finding this form is decidable — we simply normalize all of the constructor components and replace all occurrences of $T(\mathsf{Int})$ with int and all occurrences of $T(\mathsf{Arrow}(\mu_1, \mu_2))$ with $T(\mu_1) \to T(\mu_2)$. Furthermore, this normalization process preserves type equivalence.

Lemma 4.1.27 *If $\Delta \vdash \sigma$, then $\Delta \vdash \sigma \equiv NF(\sigma)$.*

Proof (sketch): Follows from confluence and strong normalization of constructors.
□

In the absence of `typerec`, a normal typing derivation is simply a derivation where we interleave uses of the non-**equiv** rules and a single use of the **equiv** rule to normalize the resulting type. However, for uses of `typerec`, we need additional uses of **equiv** to "undo" the normalization for the inductive cases.

Definition 4.1.28 (Normal Derivations) *A typing derivation $\mathcal{D}$ of the normal judgment $\Delta; \Gamma \vdash e : NF(\sigma)$ is in normal form iff $\mathcal{D}$ ends in an application of the* **equiv** *rule,*

$$(\textbf{equiv})\ \frac{(\textbf{R})\ \dfrac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Delta; \Gamma \vdash e : \sigma} \qquad \Delta \vdash \sigma \equiv NF(\sigma)}{\Delta; \Gamma \vdash e : NF(\sigma)}$$

and:

1. *the rule* **R** *is an axiom, or*
2. **R** *is neither* **equiv** *nor* **trec**, *and $\mathcal{D}_1, \cdots, \mathcal{D}_n$ are normal, or*
3. **R** *is a use of* **trec** *where the sub-derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are of the form:*

$$(\textbf{equiv})\ \frac{\mathcal{D}_1' \quad \Delta \vdash NF(\{\mathsf{Int}/t\}\sigma) \equiv \{\mathsf{Int}/t\}\sigma}{\Delta \vdash e_i : \{\mathsf{Int}/t\}\sigma}$$

and

$$(\textbf{equiv})\ \frac{\mathcal{D}_2' \quad \begin{array}{c}\Delta \vdash NF(\forall t_1, t_2{::}\Omega.\{t_1/t\}\sigma \to \{t_2/t\}\sigma \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma) \equiv \\ \forall t_1, t_2 :: \Omega.\{t_1/t\}\sigma \to \{t_2/t\}\sigma \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma\end{array}}{\Delta \vdash e_a : t_1, t_2.\{t_1/t\}\sigma \to \{t_2/t\}\sigma \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma}$$

and $\mathcal{D}_1'$ and $\mathcal{D}_2'$ are normal.

Theorem 4.1.29 *$\Delta; \Gamma \vdash e : \sigma$ iff there exists a normal derivation of $\Delta; \Gamma \vdash e : NF(\sigma)$.*

Proof: The "if" part is immediate: The normal derivation ends in $\Delta; \Gamma \vdash e : NF(\sigma)$. Since a type and its normal form are equivalent, a single additional application of the **equiv** rule yields a derivation of $\Delta; \Gamma \vdash e : \sigma$.

For the "only if" part, we use the following algorithm to transform the derivation of $\Delta; \Gamma \vdash e : \sigma$ into a normal derivation of $\Delta; \Gamma \vdash e : NF(\sigma)$. The algorithm is given by induction on the derivation $\mathcal{D}$ of $\Delta; \Gamma \vdash e : \sigma$.

If $\mathcal{D}$ is an axiom, then we add an additional **equiv** rule, using the fact that $\Delta \vdash NF(\sigma) \equiv \sigma$. The resulting derivation is in normal form.

If $\mathcal{D}$ ends with an application of **equiv**,

$$\textbf{(equiv)}\ \frac{\textbf{(R)}\ \dfrac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Delta; \Gamma \vdash e : \sigma'} \qquad \Delta \vdash \sigma' \equiv \sigma}{\Delta; \Gamma \vdash e : \sigma}$$

then the normal form of σ' must also be $NF(\sigma)$. Hence if $\mathcal{D}'_1$ and $\mathcal{D}'_n$ are the normal derivations of $\mathcal{D}_1$ through $\mathcal{D}_n$ respectively, the derivation

$$\textbf{(equiv)}\ \frac{\textbf{(R)}\ \dfrac{\mathcal{D}'_1 \quad \cdots \quad \mathcal{D}'_n}{\Delta; \Gamma \vdash e : \sigma''} \qquad \Delta \vdash \sigma'' \equiv NF(\sigma)}{\Delta; \Gamma \vdash e : NF(\sigma)}$$

is normal.

If $\mathcal{D}$ ends in

$$\textbf{(R)}\ \frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Delta; \Gamma \vdash e : \sigma}$$

where **R** is not **equiv**, we first transform $\mathcal{D}_1$ through $\mathcal{D}_n$ to normal derivations $\mathcal{D}'_1$ through $\mathcal{D}'_n$. I must now show that the rule **R** applies, followed by an application of **equiv**, yielding the normal judgment $\Delta; \Gamma \vdash e : NF(\sigma)$. There are five cases to consider:

case fn: $\mathcal{D}_1$ ends in $\Delta; \Gamma \uplus \{x{:}\sigma_1\} \vdash e : \sigma_2$. The normal derivation $\mathcal{D}'_1$ allows us to conclude that $\Delta; \Gamma \uplus x{:}\sigma_1 \vdash e : NF(\sigma_2)$ Applying the **fn** rule, we can conclude that $\Delta; \Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \to NF(\sigma_2)$. Since $\Delta \vdash \sigma_2 \equiv NF(\sigma_2)$, $\Delta \vdash \sigma_1 \to NF(\sigma_2) \equiv \sigma \equiv NF(\sigma)$. Hence, following **fn** with **equiv**, we can conclude that $\Delta; \Gamma \vdash \lambda x{:}\sigma_1.e : NF(\sigma)$.

case app: $\mathcal{D}_1$ ends in $\Delta; \Gamma \vdash e_1 : \sigma_1 \to \sigma_2$ and $\mathcal{D}_2$ ends in $\Delta; \Gamma \vdash e_2 : \sigma_1$. The normal derivation $\mathcal{D}'_1$ ends in $\Delta; \Gamma \vdash e_1 : NF(\sigma_1 \to \sigma_2)$. Now, $\Delta \vdash NF(\sigma_1 \to \sigma_2) \equiv NF(\sigma_1) \to NF(\sigma_2)$. Hence, $\mathcal{D}'_2$ ends in $\Delta; \Gamma \vdash e_2 : NF(\sigma_1)$, and applying the **app** rule, we can conclude that $\Delta; \Gamma \vdash e_1\, e_2 : NF(\sigma_2)$. Since $\Delta \vdash NF(\sigma_2) \equiv \sigma_2$, and $\sigma_2 = \sigma$, we can apply the **equiv** rule yielding $\Delta; \Gamma \vdash e_1\, e_2 : NF(\sigma)$.

case tfn: $\mathcal{D}_1$ ends in $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e_1 : \sigma_1$. The normal derivation $\mathcal{D}'_1$ ends in $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e_1 : NF(\sigma_1)$ Applying the **tfn** rule, we can conclude that $\Delta; \Gamma \vdash \Lambda t{::}\kappa.e_1 : \forall t{::}\kappa.NF(\sigma_1)$. Since $\Delta \uplus \{t{::}\kappa\} \vdash \sigma_1 \equiv NF(\sigma_1)$, $\Delta \vdash \forall t{::}\kappa.\sigma_1 \equiv \forall t{::}\kappa.NF(\sigma_1) \equiv NF(\forall t{::}\kappa.\sigma_1)$. Thus, applying the **equiv** rule, we can conclude that $\Delta; \Gamma \vdash \forall t{::}\kappa.e_1 : NF(\sigma)$.

case tapp: $\mathcal{D}_1$ ends in $\Delta; \Gamma \vdash e_1 : \forall t{::}\kappa.\sigma_1$, whereas the normal derivation $\mathcal{D}'_1$ ends in $\Delta; \Gamma \vdash e_1 : NF(\forall t{::}\kappa.\sigma_1)$. Now, $\Delta \vdash NF(\forall t{::}\kappa.\sigma_1) \equiv \forall t{::}\kappa.NF(\sigma_1)$. Applying the **tapp** rule, we can conclude that $\Delta; \Gamma \vdash e_1\, [\mu] : \{\mu/t\}NF(\sigma_1)$. By equivalence of types under substitution of a well-formed constructor, we can conclude that $\Delta \vdash \{\mu/t\}\sigma_1 \equiv \{\mu/t\}NF(\sigma_1)$. Furthermore, $\Delta \vdash \{\mu/t\}NF(\sigma_1) \equiv NF(\{\mu/t\}\sigma_1)$. Thus, $\Delta \vdash \sigma \equiv NF(\sigma)$ and by an application of the **equiv** rule, we can conclude that $\Delta; \Gamma \vdash e_1\,[\mu] : NF(\sigma)$.

case trec: $\mathcal{D}_1$ ends in $\Delta; \Gamma \vdash e_i : \{\mathsf{Int}/t\}\sigma_1$ and $\mathcal{D}_2$ ends in $\Delta; \Gamma \vdash e_a : \forall t_1, t_2{::}\Omega.\{t_1/t\}\sigma_1 \to \{t_2/t\}\sigma_1 \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma_1$. By induction, the normal derivations corresponding to these two derivations are $\mathcal{D}_1'$ which ends in $\Delta; \Gamma \vdash e_i : NF(\{\mathsf{Int}/t\}\sigma_1)$ and $\mathcal{D}_2'$ which ends in $\Delta; \Gamma \vdash e_a : NF(\forall t_1, t_2{::}\Omega.\{t_1/t\}\sigma_1 \to \{t_2/t\}\sigma_1 \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma_1$. Now $\Delta \vdash NF(\{\mathsf{Int}/t\}\sigma_1) \equiv \{t/\sigma_1\}$ and $\Delta \vdash NF(\forall t_1, t_2{::}\Omega.\{t_1/t\}\sigma_1 \to \{t_2/t\}\sigma_1 \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma_1) \equiv \forall t_1, t_2{::}\Omega.\{t_1/t\}\sigma_1 \to \{t_2/t\}\sigma_1 \to \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma_1$. So, applying **equiv** to $\mathcal{D}_1'$ and $\mathcal{D}_2'$, followed by an application of **trec**, we can conclude that

$$\Delta; \Gamma \vdash \texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma_1](e_i; e_a) : \{\mu/t\}\sigma_1$$

Since $\Delta \vdash \{\mu/t\}\sigma_1 \equiv NF(\{\mu/t\}\sigma_1)$, we can apply **equiv** to this derivation yielding $\Delta; \Gamma \vdash \texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma_1](e_i; e_a) : NF(\sigma)$. □

Theorem 4.1.30 *Given Δ, Γ, and e, where Γ is well-formed with respect to Δ, there is an algorithm to determine whether there exists a σ such that $\Delta; \Gamma \vdash e : \sigma$.*

Proof: By the previous theorem, the judgment $\Delta; \Gamma \vdash e : \sigma$ is derivable iff there is a normal derivation $\mathcal{D}$ of $\Delta; \Gamma \vdash e : NF(\sigma)$. I proceed by induction on e to calculate such a derivation if it exists, and to signal an error otherwise. By an examination of the typing rules, it is clear that for each case, at most one rule other than **equiv** applies.

case var: If e is a variable x, then the normal derivation is $\Delta; \Gamma \vdash x : \Gamma(x)$ followed by $\Delta; \Gamma \vdash x : NF(\Gamma(x))$. If x does not occur in Γ, then e is not well-typed.

case int: If e is an integer i, then the normal derivation is $\Delta; \Gamma \vdash x : \mathsf{int}$, followed by a reflexive use of equivalence (i.e., $\mathsf{int} \equiv \mathsf{int}$).

case fn: If e is a function $\lambda x{:}\sigma_1.e'$, then the bound variable can always be chosen via α-conversion so that it does not occur in the domain of Γ. If the free type variables of σ_1 are in Δ, then $\Gamma \uplus \{x{:}\sigma_1\}$ is well-formed with respect to Δ, else there is no derivation. By induction, there is an algorithm to calculate a normal derivation of $\Delta; \Gamma \uplus \{x{:}\sigma_1\} \vdash e' : NF(\sigma_2)$, if one exists. Given this derivation, by an application of the **fn** rule and a reflexive use of **equiv**, we can construct a normal derivation $\Delta; \Gamma \vdash \lambda x{:}\sigma_1.e' : NF(\sigma_1 \to \sigma_2)$.

case tfn: If e is $\Lambda t{::}\kappa.e'$, then the bound type variable can always be chosen via α-conversion so that it does not occur in the domain of Δ. Hence, the context $\Delta \uplus \{t{::}\kappa\}; \Gamma$ is well-formed. By induction, there is an algorithm to calculate a normal derivation of $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e' : NF(\sigma)$, if it exists. If so, applying **tfn** followed by a reflexive use of **equiv**, we can construct a normal derivation of $\Delta; \Gamma \vdash \Lambda t{::}\kappa.e' : NF(\forall t{::}\kappa.\sigma)$.

case app: If e is an application $e_1\, e_2$, then by induction, we can calculate normal derivations of $\Delta;\Gamma \vdash e_1 : NF(\sigma_a)$ and $\Delta;\Gamma \vdash e_2 : NF(\sigma_b)$. If such derivations exist, then either $NF(\sigma_a)$ is of the form $\sigma_1 \to \sigma_2$ or else not. If not, then no other rule applies, so the term is ill-formed. If so, then for **app** to apply, σ_1 must be the same as $NF(\sigma_b)$. If this holds, then applying the **app** rule, yields $NF(\sigma_2)$. Applying the **equiv** rule yields a normal derivation of $\Delta;\Gamma \vdash e_1\, e_2 : NF(\sigma_2)$.

case tapp: If e is $e'\,[\mu]$, then by induction, we can calculate a normal derivation of $\Delta;\Gamma \vdash e' : NF(\forall t{::}\kappa.\sigma)$ if it exists and signal an error otherwise. If it exists, applying **tapp** yields a derivation of $\Delta;\Gamma \vdash e'\,[\mu] : \{\mu/t\}\sigma$. Following this derivation with a use of **equiv** yields a normal derivation of $\Delta;\Gamma \vdash e'\,[\mu] : NF(\{\mu/t\}\sigma)$.

case trec: If e is $\texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma](e_i; e_a)$ then by induction, we can calculate normal derivations of $\Delta;\Gamma \vdash e_i : NF(\sigma_i)$ and $\Delta;\Gamma \vdash e_a : NF(\sigma_a)$ if such derivations exist. Since type equivalence is decidable, we can also determine if $\Delta \vdash NF(\sigma_i) \equiv \{\mathsf{Int}/t\}\sigma$ and $\Delta \uplus \{t_1{::}\Omega, t_2{::}\Omega\} \vdash NF(\sigma_a) \equiv \{\mathsf{Arrow}(t_1,t_2)/t\}\sigma$. If so, then we can apply the **equiv** rule, followed by the **trec** rule to yield a derivation of $\Delta;\Gamma \vdash \texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma](e_i; e_a) : \{\mu/t\}\sigma$. Then, with another application of the **equiv** rule, we can build a normal derivation of $\Delta;\Gamma \vdash \texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma](e_i; e_a) : NF(\{\mu/t\}\sigma)$. □

Corollary 4.1.31 (Type Checking Decidable) *There is an algorithm to determine whether there exists a* σ *such that* $\vdash e : \sigma$.

Proof: The type assignment $\emptyset$ is trivially well-formed with respect to the kind assignment $\emptyset$. Hence, by the previous theorem, we can calculate whether $\emptyset;\emptyset \vdash e : \sigma$ is derivable or not. □

4.2 λ_i^{ML} Type Soundness

In this section I prove that the type system for λ_i^{ML} is sound. My proof is a syntactic one in the style of Wright and Felleisen [130]. The basic idea is to show that every well-formed program e of type σ is a value of type σ, or else there exists a unique e' (modulo α-conversion), such that e steps to e' and e' has type σ. Consequently, no well-formed λ_i^{ML} program ever becomes "stuck". The notion of stuck computations is captured by the following definition.

Definition 4.2.1 (Stuck Constructors and Expressions) *A constructor is* stuck *if it is of one of the following forms:*

1. $u\,\mu$ *(u is not a λ-constructor).*

2. $\mathsf{Typerec}\ u\ \mathsf{of}\ (\mu_{\text{int}}; \mu_{\text{arrow}})$ *(u is not of the form* Int *or* $\mathsf{Arrow}(u_1, u_2)$*).*

An expression is stuck *if it is of one of the following forms:*

1. $v_1\, v_2$ *(v_1 is not a λ-expression).*

2. $v\,[\mu]$ *(v is not a Λ-expression).*

3. $\mathtt{typerec}\ U[\mu]\ \mathtt{of}\ [t.\sigma](u_i; u_a)$ *(μ is stuck).*

4. $\mathtt{typerec}\ u\ \mathtt{of}\ [t.\sigma](u_i; u_a)$ *(u is not of the form* Int *or* $\mathsf{Arrow}(u_1, u_2)$*).*

Lemma 4.2.2 (Unique Decomposition of Constructors) *A closed constructor μ is either a constructor value u, or else μ can be decomposed into a unique U and μ' such that $\mu = U[\mu']$ where μ' is either an instruction or is stuck.*

Proof: By induction on the structure of constructors. If μ is Int, $\mathsf{Arrow}(u_1, u_2)$, or $\lambda t{::}\kappa.\, \mu'$, then μ is a value. Hence, there are only two cases to consider.

case: Suppose μ is $\mu_1\, \mu_2$. There are three sub-cases to consider. First, if μ_1 and μ_2 are values u_1 and u_2, then the only decomposition of μ is an empty context $U = [\,]$ filled with $u_1\, u_2$. If u_1 is a λ-constructor, then μ is an instruction, else μ is stuck.

Second, if μ_1 is u_1, but μ_2 is not a value, then by induction, there is a unique U_2 and μ_2' such that $\mu_2 = U_2[\mu_2']$ and μ_2' is either stuck or else μ_2' is an instruction. Hence, the only decomposition of μ is $u_1\, U_2[\mu_2']$.

Third, if μ_1 is not a value, then by induction, there exists a unique U_1 and μ_1' such that $\mu_1 = U_1[\mu_1']$ and μ_1' is either an instruction or stuck. Hence, the only decomposition of e is $E_1[e_1']\, e_2$.

case: Suppose μ is $\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i; \mu_a)$. If μ_1 is a value u, then the only decomposition of μ is an empty context $U = [\,]$ filled with μ. If u is either Int or $\mathsf{Arrow}(u_1, u_2)$, the u is an instruction, else u is stuck.

If μ_1 is not a value, then by induction, there is a unique U_1 and μ_1' such that $\mu_1 = U_1[\mu_1']$ and μ_1' is either an instruction or stuck. Hence, the only decomposition of e is $\mathsf{Typerec}\ U_1[\mu_1']\ \mathsf{of}\ (\mu_i; \mu_a)$. □

Lemma 4.2.3 (Unique Decomposition of Expressions) *A closed expression e is either a value v, or else e can be decomposed into a unique E and e' such that $e = E[e']$ where e' is either an instruction or is stuck.*

Proof: By induction on the structure of expressions. The argument is similar to the one for constructors. □

Lemma 4.2.4 (Determinacy) *For any closed expression e, there is at most one value v such that $e \Downarrow v$.*

Proof: By unique decomposition, if e is not a value, there is at most one E and I such that $e = E[e_1]$ where e_1 is an instruction. Since each instruction has at most one rewriting rule, there is at most one e' such that $e \longmapsto e'$. □

The following lemma allows us to take advantage of some of the properties I proved about constructor rewriting in the context of constructor evaluation.

Lemma 4.2.5 *If $\mu \longmapsto \mu'$, then $\mu \longrightarrow \mu'$.*

Proof: By the fact that U contexts are a subset of C contexts (see Section 4.1.1), and the three evaluation rules of constructors correspond precisely to β, **t1**, and **t2**, respectively. □

Corollary 4.2.6

1. *If $\vdash \mu :: \kappa$ and $\mu \longmapsto \mu'$, then $\vdash \mu' :: \kappa$.*
2. *If $\vdash \mu :: \kappa$ and $\mu \longmapsto \mu'$, then $\vdash \mu \equiv \mu' :: \kappa$.*
3. *If $\Delta \uplus \{t{::}\kappa\} \vdash \sigma$ and $\Delta \vdash \mu :: \kappa$, then $\Delta \vdash \{\mu/t\}\sigma$.*
4. *If $\Delta \uplus \{t{::}\kappa\} \vdash \sigma$ and $\Delta \vdash \mu \equiv \mu' :: \kappa$, then $\Delta \vdash \{\mu/t\}\sigma \equiv \{\mu'/t\}\sigma$.*

Lemma 4.2.7 (Constructor Substitution) *If $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e : \sigma$, and $\Delta \vdash \mu :: \kappa$, then $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}e : \{\mu/t\}\sigma$.*

Proof: By induction on the normal derivation of $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e : NF(\sigma)$. In each case, we back up the derivation one step to the application of the non-**equiv** rule.

case var: We have $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash x : \Gamma(x)$. Thus, $\Delta; \{\mu/t\}\Gamma \vdash x : \{\mu/t\}(\Gamma(x))$.

case int: We have $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash i : \mathsf{int}$. Thus, $\Delta; \{\mu/t\}\Gamma \vdash i : \{\mu/t\}\mathsf{int}$.

case fn: We have $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \to \sigma_2$. By induction, $\Delta; \{\mu/t\}(\Gamma \uplus \{x{:}\sigma_1\}) \vdash \{\mu/t\}e : \{\mu/t\}\sigma_2$. Thus, $\Delta; \{\mu/t\}\Gamma \uplus \{x{:}\{\mu/t\}\sigma_1\} \vdash \{\mu/t\}e : \{\mu/t\}\sigma_2$. By the **fn** rule, $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}(\lambda x{:}\sigma_1.e) : \{\mu/t\}(\sigma_1 \to \sigma_2)$.

case tfn: We have $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash \Lambda t'{::}\kappa'.e : \forall t'{::}\kappa'.\sigma$. By induction, $\Delta \uplus \{t'{::}\kappa'\}; \{\mu/t\}\Gamma \vdash \{\mu/t\}e : \{\mu/t\}\sigma$, since t' can always be chosen distinct from t. By the **tfn** rule, $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}\Lambda t'{::}\kappa'.e : \{\mu/t\}\forall t'{::}\kappa.\sigma$.

case app: We have $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e_1\, e_2 : \sigma$. By induction, $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}e_1 : \{\mu/t\}(\sigma_1 \to \sigma)$ and $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}e_2 : \{\mu/t\}\sigma_1$. By the **app** rule, $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}(e_1\, e_2) : \{\mu/t\}\sigma$.

case tapp: We have $\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash e_1\,[\mu'] : \{\mu'/t'\}\sigma$. By induction, $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}e_1 : \{\mu/t\}(\forall t'{::}\kappa'.\sigma)$, where t' is chosen distinct from t via α-conversion. By the **tapp** rule, $\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}(e_1\,[\mu']) : \{\mu'/t'\}(\{\mu/t\}\sigma)$.

case trec: We have

$$\Delta \uplus \{t{::}\kappa\}; \Gamma \vdash \texttt{typerec}\ \mu'\ \texttt{of}\ [t'.\sigma](e_i; e_a) : \{\mu'/t'\}\sigma.$$

By induction,

$$\Delta; \{\mu/t\}\Gamma \vdash e_i : \{\mathsf{Int}/t'\}(\{\mu/t\}\sigma)$$

and

$$\begin{aligned}\Delta; \{\mu/t\}\Gamma \vdash e_a : \forall t_1, t_2{::}\Omega.\{t_1/t'\}(\{\mu/t\}\sigma) \to \\ \{t_2/t'\}(\{\mu/t\}\sigma) \to \{\mathsf{Arrow}(t_1, t_2)/t'\}(\{\mu/t\}\sigma).\end{aligned}$$

By the **trec** rule, since t' and t can always be chosen to be distinct via α-conversion,

$$\Delta; \{\mu/t\}\Gamma \vdash \{\mu/t\}(\texttt{typerec}\ \mu'\ \texttt{of}\ [t'.\sigma](e_i; e_a)) : \{(\{\mu/t\}\mu')/t'\}(\{\mu/t\}\sigma).$$

Thus,

$$\Delta; \{\mu/t\}\Gamma \vdash e : \{\mu/t\}(\{\mu'/t'\}\sigma).$$

□

Lemma 4.2.8 (Expression Substitution) *If* $\Delta; \Gamma \uplus \{x{:}\sigma_1\} \vdash e : \sigma$, *and* $\Delta; \Gamma \vdash e_1 : \sigma_1$, *then* $\Delta; \Gamma \vdash \{e_1/x\}e : \sigma$.

Proof: By induction on the normal derivation of $\Delta; \Gamma \uplus \{x{:}\sigma_1\} \vdash e : NF(\sigma)$. We simply replace all derivations of $\Delta; \Gamma \vdash x : \sigma_1$ with a copy of the derivation of $\Delta; \Gamma \vdash e_1 : \sigma_1$. The proof relies upon weakening the context $\Delta; \Gamma$ at these points to include the free variables that are in scope. □

Lemma 4.2.9 *If* $\vdash E[e] : \sigma$, *then there exists a* σ' *such that* $\vdash e : \sigma'$, *and for all* e' *such that* $\vdash e' : \sigma'$, $\vdash E[e'] : \sigma$.

Proof: By induction on the normal derivation of $\vdash E[e] : \sigma$, In each case, we back up the derivation one step to the application of the non-**equiv** rule.

If E is empty, the result holds trivially with $\sigma = \sigma'$. Otherwise, there are three cases to consider:

case app1: $E[e]$ is of the form $E_1[e]\,e_2$. By the typing rules, the normal derivation ends in a use of **app**. Hence, $\vdash E_1[e] : \sigma_1 \to \sigma$ and $\vdash e_2 : \sigma_1$. By induction, there exists a σ' such that $\vdash e : \sigma'$ and for all $\vdash e' : \sigma'$, $\vdash E_1[e'] : \sigma_1 \to \sigma$. Hence, by the **app** rule, there are derivations of $\vdash E_1[e]\,e_2 : \sigma$ and $\vdash E_1[e']\,e_2 : \sigma$.

case app2: $E[e]$ is of the form $v_1\, E_2[e]$. By the typing rules, the normal derivation ends in a use of **app**. Hence, $\vdash v_1 : \sigma_1 \to \sigma$ and $\vdash E_2[e] : \sigma_1$. By induction, there exists a σ' such that $\vdash e : \sigma'$ and for all $\vdash e' : \sigma'$, $\vdash E_2[e'] : \sigma_1$. Hence, by the **app** rule, there are derivations of $\vdash v\, E_2[e] : \sigma$ and $\vdash v\, E_2[e'] : \sigma$.

case tapp: $E[e]$ is of the form $E_1[e]\,[\mu]$. By the typing rules, the normal derivation ends in a use of **tapp**. Hence, $\vdash E_1[e] : \forall t{::}\kappa.\sigma_1$, where σ is $\{\mu/t\}\sigma_1$. By induction, there exists a σ' such that $\vdash e : \sigma'$ and for all $\vdash e' : \sigma'$, $\vdash E_1[e'] : \forall t{::}\kappa.\sigma_1$. Hence, by the **tapp** rule, there are derivations of $\vdash E_1[e]\,[\mu]$ and $\vdash E_1[e']\,[\mu]$. □

Lemma 4.2.10 (Type Preservation) *If $\vdash e_1 : \sigma$ and $e_1 \longmapsto e_2$, then $\vdash e_2 : \sigma$.*

Proof: By unique decomposition, there is a unique E and I such that $e_1 = E[I]$. Thus, $E[I] \longmapsto E[e']$ where $e_2 = E[e']$. By the previous lemma, there exists a σ' such that $\vdash I : \sigma'$, and it suffices to show that $\vdash e' : \sigma'$. There are six basic cases to consider, depending upon I. In each case, we use the normal derivation of $\vdash I : \sigma'$, backing up to the last use of a non-**equiv** rule.

case: I is $(\lambda x{:}\sigma_1.e'')\, v$ and thus e' is $\{v/x\}e''$. The only way that $\vdash I : \sigma'$ can be derived is by the **app** rule. Thus, $\vdash \lambda x{:}\sigma_1.e'' : \sigma_1 \to \sigma'$ and $\vdash v : \sigma_1$. Any normal derivation of $\vdash \lambda x{:}\sigma_1.e'' : \sigma_1 \to \sigma'$ must end with a use of **abs** followed by an **equiv**. Hence, $\emptyset; \{x{:}\sigma_1\} \vdash e'' : NF(\sigma')$. Therefore, by the Expression Substitution Lemma, $\vdash \{v/x\}e'' : NF(\sigma')$, and since each type is equivalent to its normal form, $\vdash e' : \sigma'$.

case: I is $(\Lambda t{::}\kappa.e'')[U[J]]$ and e' is $(\Lambda t{::}\kappa.e'')[U[\mu]]$, where $U[J] \longmapsto U[\mu]$. Any derivation of $\vdash I : \sigma'$ must end with the **tapp** rule. Hence, $\vdash \Lambda t{::}\kappa.e'' : \forall t{::}\kappa.\sigma_1$, $\vdash U[J] :: \kappa$, and $\sigma' = \{U[J]/t\}\sigma_1$. By Kind Preservation (lemma 4.1.5), $\vdash U[\mu] :: \kappa$. Thus, by the **tapp** rule, $\vdash (\Lambda t{::}\kappa.e'')[U[\mu]] : \{U[\mu]/t\}\sigma_1$. By equivalence of constructors under reduction, $\vdash U[J] \equiv U[\mu]$. Therefore, by equivalence of types under substitution of equivalent constructors, $\vdash \{U[J]/t\}\sigma_1 \equiv \{U[\mu]/t\}\sigma_1$. Thus, by the **equiv** typing rule, $\vdash (\Lambda t{::}\kappa.e'')[U[\mu]] : \{U[J]/t\}\sigma_1$.

case: I is $(\Lambda t{::}\kappa.e'')[u]$ and e' is $\{u/t\}e''$. The last step in the normal derivation of $\vdash I : \sigma'$ must be a use of **tapp**. Thus, $\vdash \Lambda t{::}\kappa.e'' : \forall t{::}\kappa.\sigma_1$ and $\sigma' = \{u/t\}\sigma_1$. Therefore, by the Constructor Substitution Lemma, $\vdash \{u/t\}e'' : \{u/t\}NF(\sigma_1)$. By equivalence of types under substitution of equivalent constructors, $\vdash \{u/t\}NF(\sigma_1) \equiv \{u/t\}\sigma_1 = \sigma$. Thus, by the **equiv** typing rule, $\vdash \{u/t\}e'' : \sigma$.

case: I is $\texttt{typerec}\ U[J]\ \texttt{of}\ [t.\sigma_1](e_i; e_a)$ and e' is $\texttt{typerec}\ U[\mu]\ \texttt{of}\ [t.\sigma_1](e_i; e_a)$. The last step in the normal derivation of $\vdash I : \sigma'$ must be a use of **trec**. Thus, $\vdash U[J] :: \Omega$ and by kind preservation, $\vdash U[\mu] :: \Omega$. Therefore, by **trec**, $\vdash \texttt{typerec}\ U[\mu]\ \texttt{of}\ [t.\sigma_1](e_i; e_a) : \sigma$

case: I is $\texttt{typerec}\ \mathsf{Int}\ \texttt{of}\ [t.\sigma_1](e_i; e_a)$ and e' is e_i. The last step in the normal derivation of $\vdash I : \sigma'$ must be a use of **trec**. Thus, σ' is equivalent to $\{\mathsf{Int}/t\}\sigma_1$. By the e_i typing hypothesis, $\vdash e_i : \{\mathsf{Int}/t\}\sigma_1$. Thus, $\vdash e' : \sigma$.

case: I is $\texttt{typerec}\ \mathsf{Arrow}(u_1, u_2)\ \texttt{of}\ [t.\sigma_1](e_i; e_a)$ and e' is

$$e_a[u_1][u_2](\texttt{typerec}\ u_1\ \texttt{of}\ [t.\sigma_1](e_i; e_a))(\texttt{typerec}\ u_2\ \texttt{of}\ [t.\sigma_1](e_i; e_a)).$$

The last step in the normal derivation of $\vdash I : \sigma'$ must be a use of **trec**. Thus, σ' is equivalent to $\{\mathsf{Arrow}(u_1, u_2)/t\}\sigma_1$. By the e_a typing hypothesis,

$$\vdash e_a : \forall t_1, t_2{::}\Omega.\{t_1/t\}\sigma_1 \to \{t_2/t\}\sigma_1 \to \{\mathsf{Arrow}(t_1, t_2)/t\}\sigma_1.$$

Thus,

$$\vdash e_a[u_1][u_2] : \{u_1/t\}\sigma_1 \to \{u_2/t\}\sigma_1 \to \{\mathsf{Arrow}(u_1, u_2)/t\}\sigma_1.$$

Therefore, $\vdash e' : \{\mathsf{Arrow}(u_1, u_2)/t\}\sigma_1$, and $\vdash e' : \sigma'$. □

Lemma 4.2.11 (Canonical Forms) *If $\vdash v : \sigma$ then:*

- *If $\vdash \sigma \equiv \mathsf{int}$, then $v = i$ for some integer i.*
- *If $\vdash \sigma \equiv \sigma_1 \to \sigma_2$, then v is of the form $\lambda x{:}\sigma_1.e$, for some x and e.*
- *If $\vdash \sigma \equiv \forall t{::}\kappa.\sigma'$, then v is of the form $\Lambda t{::}\kappa.e$, for some e.*

Proof: If $\vdash v : \sigma$, then there is a normal derivation of $\vdash v : NF(\sigma)$. Backing up this derivation by one rule, it is easy to see by the definition of values that only one rule applies. □

Lemma 4.2.12 (Constructor Progress) *If $\vdash \mu_1 :: \kappa$, then either μ_1 is a constructor value u or else there exists a μ_2 such that $\mu_1 \longmapsto \mu_2$.*

Proof: If μ_1 is not a value, then there is a unique decomposition into $U[\mu]$ for some context U and closed constructor μ. I argue that μ must be an instruction. Since $\vdash U[\mu] :: \kappa$, there exists a κ' such that $\vdash \mu :: \kappa'$.

If μ is of the form $\mu_1\, \mu_2$, then μ_1 and μ_2 must be values u_1 and u_2 respectively. Since $\vdash u_1\, u_2 :: \kappa'$, this must be derived using the **app** rule. Thus, there exists a κ_1 such that $\vdash u_1 :: \kappa_1 \to \kappa'$, and this must be derived via the **fn** rule. Therefore, u_1 must have the form $\lambda t{::}\kappa_1.\mu'$ and thus $u_1\, u_2 \longmapsto \{u_2/t\}u'$.

If μ is of the form $\mathsf{Typerec}\ \mu'\ \mathsf{of}\ (\mu_i; \mu_a)$, then μ' must be a value u. Since $\vdash \mu :: \kappa$, a derivation of this fact must end in a use of **trec**. Hence, $\vdash u :: \Omega$, and u is either Int or $\mathsf{Arrow}(u_1, u_2)$. In the former case, $\mu \longmapsto \mu_i$ and in the latter case, $\mu \longmapsto \mu_a[u_1][u_2](\mathsf{Typerec}\ u_1\ \mathsf{of}\ (\mu_i; \mu_a))(\mathsf{Typerec}\ u_2\ \mathsf{of}\ (\mu_i; \mu_a))$. □

Lemma 4.2.13 (Expression Progress) *If* $\vdash e_1 : \sigma$*, then either* e_1 *is a value or else there exists an* e_2 *such that* $e_1 \longmapsto e_2$*.*

Proof: If e_1 is not a value, then by Unique Decomposition, there exists an E and e such that $e_1 = E[e]$ and e is either stuck or else e is an instruction. I argue that e must be an instruction and thus there exists an e' such that $e \longmapsto e'$ and thus $E[e] \longmapsto e_2$ where $e_2 = E[e']$. Since $\vdash E[e] : \sigma$, there exists a σ' such that $\vdash e : \sigma'$.

case: If e is of the form $e_1\, e_2$ then both e_1 and e_2 must be values, v_1 and v_2. Since $\vdash v_1\, v_2 : \sigma'$, there is a normal derivation of $\vdash v_1\, v_2 : NF(\sigma')$ where the second to the last step uses the **app** rule. Hence, there is a σ_1 such that $\vdash v_1 : \sigma_1 \rightarrow \sigma'$ and $\vdash v_2 : \sigma_2$. A normal derivation of $\vdash v_1 : \sigma_1 \rightarrow \sigma'$ must have a second to last step that uses the **fn** rule. By Canonical Forms, v_1 must be of the form $\lambda x{:}\sigma_1.e''$. Therefore, $v_1\, v_2 \longmapsto \{v_2/x\}v_1$.

case: If e is of the form $e_1\,[\mu]$ then e_1 must be a value v_1. The second to the last step of a normal derivation of $\vdash v_1\,[\mu] : \sigma'$ must use the **tapp** rule. Thus, there exists some κ such that $\vdash \mu :: \kappa$ and $\vdash v_1 : \forall t{::}\kappa.\sigma_1$ where $\{\mu/t\}\sigma_1$ is equivalent to σ'. Any normal derivation of $\vdash v_1 : \forall t{::}\kappa.\sigma_1$ must have a second to last step that uses the **tfn** rule. By Canonical Forms, v_1 must be of the form $\Lambda t{::}\kappa.e''$. If μ is a constructor value u, then $v_1\,[u] \longmapsto \{u/t\}e''$. Otherwise, by constructor progress, there exists a μ'' such that $\mu \longmapsto \mu''$. Therefore, $v_1\,[\mu] \longmapsto v_1\,[U[\mu'']]$.

case: If e is of the form $\texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma_1](e_i; e_a)$, then by constructor progress μ is either a constructor value u or else there exists a μ' such that $\mu \longmapsto \mu'$ and thus $e \longmapsto \texttt{typerec}\ \mu'\ \texttt{of}\ [t.\sigma_1](e_i; e_a)$. If μ is a constructor value u, then any normal derivation of $\vdash e : \sigma'$ must end with the second to last step an application of **trec**. Hence, $\vdash u :: \Omega$. Therefore, u is either Int or $\mathsf{Arrow}(u_1, u_2)$ for some u_1 and u_2. In the former case, $e \longmapsto e_i$ and in the latter case,

$$e \longmapsto e_a[u_1][u_2](\texttt{typerec}\ u_1\ \texttt{of}\ [t.\sigma_1](e_i; e_a))(\texttt{typerec}\ u_2\ \texttt{of}\ [t.\sigma_1](e_i; e_a)).$$

□

Corollary 4.2.14 (Stuck Programs Untypeable) *If* $\vdash e : \sigma$*, then* e *is not stuck.*

Proof: If $\vdash e : \sigma$, then by progress, either e is a value or else $e \longmapsto e'$ for some e'. If $e \longmapsto e'$, then by preservation, $\vdash e' : \sigma'$. □

Corollary 4.2.15 (Soundness) *If* $\vdash e : \sigma$ *then* e *cannot become stuck.*

Proof: I argue by induction on n that if $e \longmapsto^n e'$, then e' is not stuck. For $n = 0$, the result holds by the previous corollary. Suppose $e \longmapsto^n e'$. By the induction hypothesis, e' is not stuck. Hence, e' is either a value or else $e' \longmapsto e''$. By Preservation, $\vdash e'' : \sigma$. Thus, by the previous lemma, e'' is not stuck. □

Chapter 5

Compiling with Dynamic Type Dispatch

In this chapter, I show how to compile Mini-ML to a variant of λ_i^{ML}, called λ_i^{ML}-Rep. The primary purpose of the translation is to give a simple, but compelling demonstration of the power of dynamic type dispatch. A secondary purpose is to demonstrate the methodology of type-directed compilation and present a proof of translation correctness.

I also address two real implementation issues with the translation: first, I show how to eliminate structural, polymorphic equality by using a combination of primitive equality operations and dynamic type dispatch. As a result, the target language does not need specialized support for dispatching on values. This implies that λ_i^{ML}-Rep does not need tags on values to support polymorphic equality.

Second, I flatten 1-argument functions into multiple argument functions when possible. Recall that Mini-ML, like SML, provides only 1-argument functions. We can simulate multiple arguments by passing a tuple to a function, but it is best to pass multiple arguments directly in registers since access to registers is typically faster than access to an allocated object. Therefore, if a source function takes a tuple as an argument, I translate it so that the components of the tuple are passed directly as multiple, unallocated arguments. I use dynamic type dispatch to determine whether to flatten a function that takes an argument of unknown type.

In practice, flattening function arguments yields a substantial speedup for SML code and significantly reduces allocation. Much of the improvement claimed by Shao and Appel for their implementation of Leroy-style representation analysis is due to argument flattening [110]. In particular, they reduced total execution time by 11% on average and allocation by 30% on average. I found similar performance advantages with argument flattening in the context of the TIL compiler (see Chapter 8).

After demonstrating how multi-argument functions and polymorphic equality may be implemented in λ_i^{ML}, I sketch how other language constructs, notably C-style structs,

(kinds)	κ	$::=$	$\Omega \mid \kappa_1 \to \kappa_2$
(constructors)	μ	$::=$	$t \mid \mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Unit} \mid \mathsf{Prod}(\mu_1, \mu_2) \mid \mathsf{Arrow}([\mu_1, \cdots, \mu_k], \mu) \mid$ $\lambda t{::}\kappa.\, \mu \mid \mu_1\, \mu_2 \mid \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)$
(types)	σ	$::=$	$T(\mu) \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{unit} \mid \langle \sigma_1 \times \sigma_2 \rangle \mid [\sigma_1, \cdots, \sigma_k] \to \sigma_2 \mid \forall t{::}\kappa.\sigma$
(expressions)	e	$::=$	$x \mid i \mid f \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \lambda[x{:}\sigma_1, \cdots, x_k{:}\sigma_k].\, e \mid$ $e\,[e_1, \cdots, e_k] \mid \Lambda t{::}\kappa.e \mid e[\mu] \mid$ $\texttt{eqint}(e_1, e_2) \mid \texttt{eqfloat}(e_1, e_2) \mid \texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid$ $\texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma](e_i; e_f; e_u; e_p; e_a)$

Figure 5.1: Syntax of λ_i^{ML}-Rep

Haskell-style type classes, and polymorphic communication primitives, can be coded using dynamic type dispatch. The rest of this chapter proceeds as follows: In Section 5.1, I define the target language, λ_i^{ML}-Rep. In Section 5.2, I define a type-directed translation from Mini-ML to λ_i^{ML}-Rep that eliminates equality and flattens function arguments. In Section 5.3 I prove the correctness of this translation. Finally, In Section 5.4, I demonstrate how other constructs may be implemented through dynamic type dispatch.

5.1 The Target Language: λ_i^{ML}-Rep

The syntactic classes of the target language, λ_i^{ML}-Rep, are given in Figure 5.1. λ_i^{ML}-Rep extends λ_i^{ML} by adding floats, products, and k-argument functions (for some fixed, but arbitrary k.) The constructor level reflects these changes by the addition of Float, Unit, $\mathsf{Prod}(\mu_1, \mu_2)$, and $\mathsf{Arrow}([\mu_1, \cdots, \mu_k], \mu)$ constructors, and by the addition of arms within $\mathsf{Typerec}$ and `typerec` corresponding to these constructors. In addition, λ_i^{ML}-Rep provides primitive equality functions for integer and floating point values (`eqint` and `eqfloat`) as well as an `if0` construct. However, λ_i^{ML}-Rep does not provide a polymorphic equality operator.

The values, evaluation contexts, and rewriting rules of λ_i^{ML}-Rep are essentially the same as for λ_i^{ML} (see Section 3.3), with the addition of the product operations, the equality operations, and `if0`, so I omit these details. The added constructor and term formation rules for λ_i^{ML}-Rep are given in Figures 5.2 and 5.3 respectively.

Technically, all λ_i^{ML}-Rep functions have k arguments, but I use $\lambda[x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n].\, e$ where $n \leq k$ to represent the term:

$$\lambda[x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n, x_{n+1}{:}\mathsf{unit}, \cdots, x_k{:}\mathsf{unit}].\, e$$

when $x_{n+1}, \cdots, x_k$ do not occur free in e. Similarly, I write $e\,[e_1, \cdots, e_n]$ where $n \leq k$ to represent the term $e\,[e_1, \cdots, e_n, \langle\rangle_{n+1}, \cdots, \langle\rangle_k]$. In the degenerate case where $n = 1$, I drop the brackets entirely and simply write $\lambda x_1{:}\sigma_1.\, e$ and $e\, e_1$. I use similar abbreviations for arrow constructors and types.

When convenient, I use ML-style pattern matching to define a constructor involving Typerec or a term involving `typerec`. For instance, instead of writing

$$\begin{array}{rcl} \mathsf{F} & = & \lambda t{::}\kappa.\mathsf{Typerec}\ t\ \mathsf{of} \\ & & \quad (\mu_i; \mu_f; \mu_u; \\ & & \quad \lambda t_1{::}\Omega.\lambda t_2{::}\Omega.\lambda t_1'{::}\kappa.t_2'{::}\kappa.\mu_p \\ & & \quad \lambda t_1{::}\Omega.\cdots.\lambda t_k.\lambda t{::}\Omega.\lambda t_1'{::}\kappa.\cdots.\lambda t_k'{::}\kappa, t{::}\kappa.\mu_a) \end{array}$$

I write:

$$\begin{array}{rcl} \mathsf{F}[\mathsf{Int}] & = & \mu_i \\ \mathsf{F}[\mathsf{Float}] & = & \mu_f \\ \mathsf{F}[\mathsf{Unit}] & = & \mu_u \\ \mathsf{F}[\mathsf{Prod}(t_1, t_2)] & = & \{\mathsf{F}[t_1]/t_1', \mathsf{F}[t_2]/t_2'\}\mu_p \\ \mathsf{F}[\mathsf{Arrow}([t_1, \cdots, t_k], t)] & = & \{\mathsf{F}[t_1]/t_1', \cdots, \mathsf{F}[t_k]/t_k', \mathsf{F}[t]/t'\}\mu_a \end{array}$$

As in SML, I use an underscore ("_") to represent a wildcard match.

I also use the derived form Typecase for an application of Typerec where the inductive cases are unused. Hence, the pattern matching constructor

$$\begin{array}{rcl} \mathsf{F}[\mathsf{Int}] & = & \mu_i \\ \mathsf{F}[\mathsf{Float}] & = & \mu_f \\ \mathsf{F}[\mathsf{Unit}] & = & \mu_u \\ \mathsf{F}[\mathsf{Prod}(t_1, t_2)] & = & \{\mathsf{F}[t_1]/t_1', \mathsf{F}[t_2]/t_2'\}\mu_p \\ \mathsf{F}[\mathsf{Arrow}([t_1, \cdots, t_k], t)] & = & \{\mathsf{F}[t_1]/t_1', \cdots, \mathsf{F}[t_k]/t_k', \mathsf{F}[t]/t'\}\mu_a \end{array}$$

where t_1' and t_2' do not occur free in μ_p and $t_1', \cdots, t_k', t'$ do not occur free in μ_a may be written using Typecase as follows:

$$\begin{array}{l} \mathsf{Typecase}\ \mu\ \mathsf{of} \\ \quad \mathsf{Int} \Rightarrow \mu_i \\ \mid\ \mathsf{Float} \Rightarrow \mu_f \\ \mid\ \mathsf{Unit} \Rightarrow \mu_u \\ \mid\ \mathsf{Prod}(t_1, t_2) \Rightarrow \mu_p \\ \mid\ \mathsf{Arrow}([t_1, \cdots, t_k], t_2) \Rightarrow \mu_a \end{array}$$

Similarly, I use a derived `typecase` expression form for instances of `typerec` where the inductive cases cases are unused.

$$(\textbf{unit})\ \Delta \vdash \mathsf{Unit} :: \Omega \qquad\qquad (\textbf{prod})\ \frac{\Delta \vdash \mu_1 :: \Omega \qquad \Delta \vdash \mu_2 :: \Omega}{\Delta \vdash \mathsf{Prod}(\mu_1, \mu_2) :: \Omega}$$

$$(\textbf{arrow})\ \frac{\begin{array}{c}\Delta \vdash \mu_1 :: \Omega \quad \cdots \quad \Delta \vdash \mu_k :: \Omega \\ \Delta \vdash \mu :: \Omega\end{array}}{\Delta \vdash \mathsf{Arrow}([\mu_1, \cdots, \mu_k], \mu) :: \Omega}$$

$$(\textbf{trec})\ \frac{\begin{array}{c}\Delta \vdash \mu :: \Omega \quad \Delta \vdash \mu_i :: \kappa \\ \Delta \vdash \mu_f :: \kappa \quad \Delta \vdash \mu_u :: \kappa \\ \Delta \vdash \mu_p :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \Delta \vdash \mu_{\text{arrow}} :: \Omega_1 \to \cdots \to \Omega_k \to \Omega \to \kappa_1 \to \cdots \to \kappa_k \to \kappa \to \kappa\end{array}}{\Delta \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a) :: \kappa} \quad (\kappa_1, \cdots, \kappa_k = \kappa)$$

Figure 5.2: Added Constructor Formation Rules for λ_i^{ML}-Rep

5.2 Compiling Mini-ML to λ_i^{ML}-Rep

With the definitions of the source and target languages in place, I can define a translation from Mini-ML to λ_i^{ML}-Rep. In this section, I carefully develop such a translation, concentrating first on a translation from Mini-ML types to λ_i^{ML}-Rep constructors. Next, I develop a term translation and show that it respects the type translation. Finally, I prove that the translation is correct by establishing a suitable family of simulation relations and by showing that a well-formed Mini-ML expression is appropriately simulated by its λ_i^{ML}-Rep translation.

5.2.1 Translation of Types

I translate Mini-ML monotypes to λ_i^{ML}-Rep constructors via the function $|\tau|$, which is defined by induction on τ as follows:

$$\begin{array}{rcl}
|t| & = & t \\
|\mathsf{int}| & = & \mathsf{Int} \\
|\mathsf{float}| & = & \mathsf{Float} \\
|\mathsf{unit}| & = & \mathsf{Unit} \\
|\langle \tau_1 \times \tau_2 \rangle| & = & \langle |\tau_1| \times |\tau_2| \rangle \\
|\tau_1 \to \tau_2| & = & \mathsf{Vararg}\ |\tau_1|\ |\tau_2|
\end{array}$$

$$(\textbf{prod})\ \frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \quad \Gamma;\Delta \vdash e_2}{\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \langle \sigma_1 \times \sigma_2\rangle}$$

$$(\textbf{proj})\ \frac{\Delta;\Gamma \vdash e : \langle \sigma_1 \times \sigma_2\rangle}{\Delta;\Gamma \vdash \pi_i\, e : \sigma_i} \quad (n = 1, 2)$$

$$(\textbf{eqi})\ \frac{\Delta;\Gamma \vdash e_1 : \mathsf{int} \quad \Delta;\Gamma \vdash e_2 : \mathsf{int}}{\Delta;\Gamma \vdash \mathtt{eqint}(e_1, e_2) : \mathsf{int}} \qquad (\textbf{eqf})\ \frac{\Delta;\Gamma \vdash e_1 : \mathsf{float} \quad \Delta;\Gamma \vdash e_2 : \mathsf{float}}{\Delta;\Gamma \vdash \mathtt{eqfloat}(e_1, e_2) : \mathsf{int}}$$

$$(\textbf{if0})\ \frac{\Delta;\Gamma \vdash e_1 : \mathsf{int} \quad \Delta;\Gamma \vdash e_2 : \sigma \quad \Delta;\Gamma \vdash e_3 : \sigma}{\Delta;\Gamma \vdash \mathtt{if0}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \sigma}$$

$$(\textbf{abs})\ \frac{\Delta;\Gamma \uplus \{x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k\} \vdash e : \sigma}{\Delta;\Gamma \vdash \lambda[x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].\, e : [\sigma_1, \cdots, \sigma_k] \to \sigma}$$

$$(\textbf{app})\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e : [\sigma_1, \cdots, \sigma_k] \to \sigma \\ \Delta;\Gamma \vdash e_1 : \sigma_1 \quad \cdots \quad \Delta;\Gamma \vdash e_k : \sigma_k\end{array}}{\Delta;\Gamma \vdash e\,[e_1, \cdots, e_k] : \sigma_k}$$

$$(\textbf{trec})\ \frac{\begin{array}{c}\Delta \vdash \mu :: \Omega \qquad \Delta;\Gamma \vdash e_i : \{\mathsf{Int}/t\}\sigma \\ \Delta;\Gamma \vdash e_f : \{\mathsf{Float}/t\}\sigma \qquad \Delta;\Gamma \vdash e_u : \{\mathsf{Unit}/t\}\sigma \\ \Delta;\Gamma \vdash e_p : \forall t_1{::}\Omega.\forall t_2{::}\Omega.[\{t_1/t\}\sigma] \to [\{t_2/t\}\sigma] \to \{\mathsf{Prod}(t_1, t_2)/t\}\sigma \\ \Delta;\Gamma \vdash e_a : \forall t_1{::}\Omega. \cdots .\forall t_k{::}\Omega.\forall t'{::}\Omega. \\ \quad [\{t_1/t\}\sigma] \to \cdots \to [\{t_k/t\}\sigma] \to [\{t'/t\}\sigma] \to \\ \{\mathsf{Arrow}([t_1, \cdots, t_k], t')/t\}\sigma\end{array}}{\Delta;\Gamma \vdash \mathtt{typerec}\ \mu\ \mathtt{of}\ [t.\sigma](e_i; e_f; e_u; e_p; e_a) : \{\mu/t\}\sigma}$$

Figure 5.3: Added Term Formation Rules for λ_i^{ML}-Rep

The translation maps each type to its corresponding constructor except for arrow types: An arrow type whose domain is a tuple is flattened into an arrow constructor with multiple arguments by the Vararg constructor function. Vararg is defined using $\mathsf{Typecase}$ as follows:

$$
\begin{array}{rcl}
\mathsf{Vararg} & = & \lambda t{::}\Omega.\lambda t'{::}\Omega. \\
& & \quad \mathsf{Typecase}\ t\ \mathsf{of} \\
& & \qquad \mathsf{Prod}(t_1, t_2) \Rightarrow \mathsf{Arrow}([t_1, t_2], t') \\
& & \quad\ \mid \_ \Rightarrow \mathsf{Arrow}(t, t'),
\end{array}
$$

and has the property that, if $\tau = \langle \tau_1 \times \tau_2 \rangle$, then

$$\mathsf{Vararg}\ |\tau|\ |\tau'| \equiv \mathsf{Arrow}([|\tau_1|, |\tau_2|], |\tau'|).$$

Alternatively, if τ is not a product (and not a variable), then Vararg does not flatten the domain. For instance,

$$\mathsf{Vararg}\ \mathsf{Int}\ |\tau'| \equiv \mathsf{Arrow}(\mathsf{Int}, |\tau'|).$$

In effect, Vararg *reifies* the type translation for arrow types as a function at the constructor level.

The type translation is extended to map source type schemes to target types, source type assignments to target type assignments, and source kind assignments to target kind assignments as follows:

$$
\begin{array}{rcl}
|\forall t_1, \cdots, t_n.\tau| & = & \forall t_1{::}\Omega. \cdots .\forall t_n{::}\Omega.T(|\tau|) \\[1em]
|\Gamma| & = & \{x{:}|\Gamma(x)| \mid x \in Dom(\Gamma)\} \\[1em]
|\Delta| & = & \{t{::}\Omega \mid t \in Dom(\Delta)\}
\end{array}
$$

This type translation has the very important property that it commutes with substitution. This is in stark contrast to any of the coercion-based approaches to polymorphism, where this property does not hold and a term-level coercion must be used to mitigate the mismatch. In some sense, my type translation is "self-correcting" when I perform substitution, because the computation of an arrow type, whose domain is unknown, is delayed until the type is apparent.

Lemma 5.2.1 $|\{\tau'/t\}\tau| \equiv \{|\tau'|/t\}|\tau|$.

Proof: By induction on τ. □

5.2.2 Translation of Terms

I specify the translation of Mini-ML expressions as a deductive system using judgments of the form $\Delta; \Gamma \vdash e : \sigma \Rightarrow e'$ where $\Delta; \Gamma \vdash e : \sigma$ is a Mini-ML typing judgment and e' is the λ_i^{ML}-Rep translation of e. The axioms and inference rules that allow us to conclude this judgment are given in Figure 5.4.

Much of the translation is straightforward: The translation of variables, integers, and floating point values is the identity; the translation of `if0` and π expressions is obtained by simply translating the component expressions. In the following subsections, I present the translation of equality, functions, application, type abstractions, and type application in detail.

5.2.3 Translation of Equality

An equality operation is translated according to the following rule:

$$\frac{\Delta; \Gamma \vdash e_1 : \tau \Rightarrow e_1' \qquad \Delta; \Gamma \vdash e_2 : \tau \Rightarrow e_2'}{\Delta; \Gamma \vdash \mathsf{eq}(e_1, e_2) : \mathsf{int} \Rightarrow \mathsf{peq}[|\tau|][e_1', e_2']}$$

The translation uses an auxiliary function, peq, that can be coded in the target language using `typerec`. Here, I use the pattern-matching syntax to define such a function:

$$\begin{array}{lcl}
\mathsf{peq}[\mathsf{Int}] & = & \lambda[x_1{:}\mathsf{int}, x_2{:}\mathsf{int}].\, \mathtt{eqint}(x_1, x_2) \\
\mathsf{peq}[\mathsf{Float}] & = & \lambda[x_1{:}\mathsf{float}, x_2{:}\mathsf{float}].\, \mathtt{eqfloat}(x_1, x_2) \\
\mathsf{peq}[\mathsf{Unit}] & = & \lambda[x_1{:}\mathsf{unit}, x_2{:}\mathsf{unit}].\, 1 \\
\mathsf{peq}[\mathsf{Prod}(t_a, t_b)] & = & \lambda\, [x_1{:}T(\mathsf{Prod}(t_a, t_b)), x_2{:}T(\mathsf{Prod}(t_a, t_b))]. \\
 & & \quad \mathtt{if0}\ \mathsf{peq}[t_a][\pi_1\, x_1, \pi_1\, x_2]\ \mathtt{then}\ 0\ \mathtt{else}\ \mathsf{peq}[t_b][\pi_2\, x_1, \pi_2\, x_2] \\
\mathsf{peq}[t] & = & \lambda\, [x_1{:}T(t), x_2{:}T(t)].\, 0
\end{array}$$

Operationally, peq takes a constructor as an argument and selects the appropriate comparison function according to that constructor. For a product $\mathsf{Prod}(t_a, t_b)$, the appropriate function is constructed by using the inductive arguments $\mathsf{peq}[t_a]$ and $\mathsf{peq}[t_b]$ to compare the components of the product.

Expanding the pattern matching abbreviation to a `typerec` yields:

$$\Lambda t{::}\Omega.\mathtt{typerec}\ t\ \mathtt{of}\ [t.\sigma](e_i; e_f; e_u; e_p; e_a)$$

$$(\mathbf{var})\ \ \Delta;\Gamma \uplus \{x{:}\sigma\} \vdash x : \sigma \Rightarrow x \qquad\qquad (\mathbf{unit})\ \ \Delta;\Gamma \vdash \langle\rangle : \mathsf{unit} \Rightarrow \langle\rangle$$

$$(\mathbf{int})\ \ \Delta;\Gamma \vdash i : \mathsf{int} \Rightarrow i \qquad\qquad (\mathbf{float})\ \ \Delta;\Gamma \vdash f : \mathsf{float} \Rightarrow f$$

$$(\mathbf{eq})\ \ \frac{\Delta;\Gamma \vdash e_1 : \tau \Rightarrow e_1' \quad \Delta;\Gamma \vdash e_2 : \tau \Rightarrow e_2'}{\Delta;\Gamma \vdash \mathtt{eq}(e_1, e_2) : \mathsf{int} \Rightarrow \mathtt{peq}[|\tau|][e_1', e_2']}$$

$$(\mathbf{if0})\ \ \frac{\Delta;\Gamma \vdash e_1 : \mathsf{int} \Rightarrow e_1' \quad \Delta;\Gamma \vdash e_2 : \tau \Rightarrow e_2' \quad \Delta;\Gamma \vdash e_3 : \tau \Rightarrow e_3'}{\Delta;\Gamma \vdash \mathtt{if0}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\, e_3 : \tau \Rightarrow \mathtt{if0}\ e_1'\ \mathtt{then}\ e_2'\ \mathtt{else}\, e_3'}$$

$$(\mathbf{pair})\ \ \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \Rightarrow e_1' \quad \Delta;\Gamma \vdash e_2 : \tau_2 \Rightarrow e_n'}{\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \langle \tau_1 \times \tau_2\rangle \Rightarrow \langle e_1', e_2'\rangle}$$

$$(\mathbf{proj})\ \ \frac{\Delta;\Gamma \vdash e : \langle \tau_1 \times \tau_2\rangle \Rightarrow e'}{\Delta;\Gamma \vdash \pi_i\, e : \tau_i \Rightarrow \pi_i\, e'} \quad (i = 1, 2)$$

$$(\mathbf{abs})\ \ \frac{\Delta;\Gamma \uplus \{x{:}\tau_1\} \vdash e : \tau_2 \Rightarrow e'}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2 \Rightarrow \mathtt{vararg}[|\tau_1|][|\tau_2|](\lambda x{:}T(|\tau_1|).\, e')}$$

$$(\mathbf{app})\ \ \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \Rightarrow e_1' \quad \Delta;\Gamma \vdash e_2 : \tau_1 \Rightarrow e_2'}{\Delta;\Gamma \vdash e_1\, e_2 : \tau_2 \Rightarrow (\mathtt{onearg}[|\tau_1|][|\tau_2|]\, e_1')\, e_2'}$$

$$(\mathbf{def})\ \ \frac{\Delta;\Gamma \vdash v : \sigma \Rightarrow v' \quad \Delta;\Gamma \uplus \{x{:}\sigma\} \vdash e : \tau \Rightarrow e'}{\Delta;\Gamma \vdash \mathtt{def}\, x{:}\sigma = v\ \mathtt{in}\ e : \tau \Rightarrow \mathtt{let}\, x{:}|\sigma| = v'\ \mathtt{in}\ e'}$$

$$(\mathbf{tapp})\ \ \frac{\begin{array}{c}\Delta \vdash \tau_1 \quad \cdots \quad \Delta \vdash \tau_n \\ \Delta;\Gamma \vdash v : \forall t_1, \cdots, t_n.\tau \Rightarrow v'\end{array}}{\Delta;\Gamma \vdash v[\tau_1, \cdots, \tau_n] : \{\tau_1/t_1, \cdots, \tau_n/t_n\}\tau \Rightarrow v'[|\tau_1|] \cdots [|\tau_n|]}$$

$$(\mathbf{tabs})\ \ \frac{\Delta \uplus \{t_1, \cdots, t_n\};\Gamma \vdash e : \tau \Rightarrow e'}{\Delta;\Gamma \vdash \Lambda t_1, \cdots, t_n.e : \forall t_1, \cdots, t_n.\tau \Rightarrow \Lambda t_1{::}\Omega. \cdots .\Lambda t_n{::}\Omega.e'}$$

Figure 5.4: Translation from Mini-ML to λ_i^{ML}-Rep

where (eliding some kind and type information)

$$\begin{array}{rcl}
\sigma & = & [T(t), T(t)] \to \mathsf{int} \\
e_i & = & \lambda[x_1{:}\mathsf{int}, x_2{:}\mathsf{int}].\, \mathtt{eqint}(x_1, x_2) \\
e_f & = & \lambda[x_1{:}\mathsf{float}, x_2{:}\mathsf{float}].\, \mathtt{eqfloat}(x_1, x_2) \\
e_u & = & \lambda[x_1{:}\mathsf{unit}, x_2{:}\mathsf{unit}].\, 1 \\
e_p & = & \Lambda t_a.\Lambda t_b.\lambda \mathsf{peq}_a.\lambda \mathsf{peq}_b.\lambda[x_1{:}T(\mathsf{Prod}(t_a,t_b)), x_2{:}T(\mathsf{Prod}(t_a,t_b))]. \\
& & \qquad \mathtt{if0}\ \mathsf{peq}_a[\pi_1\, x_1, \pi_1\, x_2]\ \mathtt{then}\ 0\ \mathtt{else}\, \mathsf{peq}_b[\pi_2\, x_1, \pi_2\, x_2] \\
e_a & = & \Lambda t_1.\cdots\Lambda t_k.\Lambda t'.\lambda \mathsf{peq}_1.\cdots\lambda \mathsf{peq}_2.\lambda \mathsf{peq}'. \\
& & \qquad \lambda[x_1{:}T(\mathsf{Arrow}([t_1,\cdots,t_k],t')), x_2{:}T(\mathsf{Arrow}([t_1,\cdots,t_k],t'))].0
\end{array}$$

From this definition, it is easy to verify that

$$\vdash \mathsf{peq} : \forall t{::}\Omega.[T(t), T(t)] \to \mathsf{int}.$$

The derivation proceeds as follows: By the **tabs** rule (see Figure 5.4), it suffices to show

$$\{t{::}\Omega\};\emptyset \vdash \mathtt{typerec}\ t\ \mathtt{of}\ [t.\sigma](e_i; e_f; e_u; e_p; e_a) : \sigma.$$

This follows if I can derive the preconditions of the **trec** rule for λ_i^{ML}-Rep (see Figure 5.3). For instance, I must show that

$$\{t{::}\Omega\};\emptyset \vdash e_i : \{\mathsf{Int}/t\}\sigma,$$

which follows from the derivation below:

$$\dfrac{\dfrac{\{t{::}\Omega\};\{x_1{:}\mathsf{int}, x_2{:}\mathsf{int}\} \vdash x_1 : \mathsf{int} \quad \{t{::}\Omega\};\{x_1{:}\mathsf{int}, x_2{:}\mathsf{int}\} \vdash x_2 : \mathsf{int}}{\{t{::}\Omega\};\{x_1{:}\mathsf{int}, x_2{:}\mathsf{int}\} \vdash \mathtt{eqint}(x_1, x_2) : \mathsf{int}}}{\{t{::}\Omega\};\emptyset \vdash \lambda[x_1{:}\mathsf{int}, x_2{:}\mathsf{int}].\mathtt{eqint}(x_1, x_2) : \{\mathsf{Int}/t\}([T(t), T(t)] \to \mathsf{int})}$$

The other cases follow in a similar manner.

Intuitively, peq implements the first five rewriting rules of the dynamic semantics of Mini-ML (see Figure 2.3), but it does so by dispatching on its constructor argument instead of the shape of its value arguments.

5.2.4 Translation of Functions

There are three cases to consider when translating a λ-expression:

1. the argument type is known to be a tuple;

2. the argument type is int, float, unit, or an arrow type;

3. the argument type is a type variable.

In the first case, the argument is a tuple. I need to produce a function that takes the components of the tuple directly as arguments. I translate the body of the function under the assumption that the argument was passed as a tuple. Then, I abstract the arguments appropriately. However, before executing the body of the function, I allocate a tuple and bind it to the original parameter, x.

$$\frac{\Delta; \Gamma \uplus \{x{:}\langle\tau_1 \times \tau_2\rangle\} \vdash e : \tau' \Rightarrow e'}{\begin{array}{c}\Delta; \Gamma \vdash \lambda x{:}\langle\tau_1 \times \tau_2\rangle.\, e : \langle\tau_1 \times \tau_2\rangle \to \tau' \Rightarrow \\ \lambda[x_1{:}T(|\tau_1|), x_2{:}T(|\tau_2|)].\, \mathtt{let}\, x{:}T(|\langle\tau_1 \times \tau_2\rangle|) = \langle x_1, x_2\rangle \;\mathtt{in}\; e'\end{array}} \quad (x_1, x_2 \notin \mathit{Dom}(\Gamma))$$

It is easy for an optimizer to replace projections $\pi_i\, x$ within the translated body of the function with the appropriate argument, x_i. When the tuple is used only to simulate multiple arguments, the variable x will occur only within such projections. Hence, all occurrences of x will be eliminated by the optimizer, and the binding of the tuple to x will become "dead" and can be eliminated altogether.

I leave these optimizations out of the translation for two reasons: first, such optimizations make reasoning about the underlying translation more difficult. Second, the optimizations (projection elimination and dead code elimination) are generally useful and could be applied after other passes in the compiler. Hence, for the sake of modularity it is best to leave these transformations as separate passes over the target code.

In the second case, the argument is a non-tuple and a non-variable. No flattening need occur and the translation is straightforward:

$$\frac{\Delta; \Gamma \uplus \{x{:}\tau\} \vdash e : \tau' \Rightarrow e'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau' \Rightarrow \lambda x{:}T(|\tau|).\, e'}$$

In the third case, the argument type of the function is a type variable (t). If this variable is instantiated with a tuple type, then the function should be flattened; otherwise, the function should not be flattened. I use a term-level `typecase` to decide which calling convention to use. To avoid duplicating the function body for each case, I borrow an idea from the coercion-based approaches: Pick one calling convention, compile the function using this convention, and for each case, calculate a coercion from the expected to the actual calling convention. For instance, I might compile the function as if there was one argument of type t and then use `typecase` to calculate the proper coercion to multiple arguments, depending on the instantiation of t. This leads to the following translation:

$$\frac{\Delta; \Gamma \uplus \{x{:}t\} \vdash e : \tau_2 \Rightarrow e'}{\Delta; \Gamma \vdash \lambda x{:}t.\, e : t \to \tau_2 \Rightarrow \mathtt{vararg}[t][|\tau_2|](\lambda x{:}T(t).\, e')}$$

where the term `vararg` is defined as follows:

$$
\begin{array}{lcl}
\texttt{vararg} & = & \Lambda t{::}\Omega.t'{::}\Omega. \\
& & \quad \texttt{typecase}\ t\ \mathsf{of} \\
& & \qquad \mathsf{Prod}(t_1, t_2) \texttt{ => } \\
& & \qquad\ \lambda[x{:}[T(\mathsf{Prod}(t_1, t_2))] \to T(t')].\lambda[x_1{:}T(t_1), x_2{:}T(t_2)].x\,[\langle x_1, x_2\rangle] \\
& & \quad\ \mid\ \_\texttt{=>}\ \lambda[x{:}[T(t)] \to T(t')].x
\end{array}
$$

Expanding the pattern matching `typecase` to a formal `typerec` yields

$$\texttt{typerec}\ t\ \mathsf{of}\ [t.\sigma](e_0[\mathsf{Int}]; e_0[\mathsf{Float}]; e_0[\mathsf{Unit}]; e_p; e_a),$$

where (eliding some kind and type annotations)

$$
\begin{array}{rl}
\sigma = & [[T(t)] \to T(t')] \to T(\mathsf{Vararg}\ t\ t') \\
e_0 = & \Lambda t''{::}\Omega.\lambda[[x{:}T(t'')] \to T(t')].x \\
e_a = & \Lambda t_1.\cdots.\Lambda t_k.\Lambda t''.\lambda[x_1].\cdots.\lambda[x_k].\lambda[x''].\lambda[x{:}[T(\mathsf{Arrow}([t_1, \cdots, t_k], t''))] \to T(t')].x \\
e_p = & \Lambda t_1.\Lambda t_2.\lambda[x_1'].\lambda[x_2'].\lambda[x{:}[T(\mathsf{Prod}(t_1, t_2))] \to T(t')].\lambda[x_1{:}T(t_1), x_2{:}T(t_2)].x\,[\langle x_1, x_2\rangle].
\end{array}
$$

Notice that the inductive arguments for e_a ($x_1, \cdots, x_k$ and x'') and e_p (x_1' and x_2') are unused. It is straightforward to show that the expansion of `vararg` yields a well-formed term with type $\forall t{::}\Omega.\forall t'{::}\Omega.\sigma$.

Lemma 5.2.2 $\vdash \texttt{vararg} : \forall t{::}\Omega.\forall t'{::}\Omega.(T(t) \to T(t')) \to (T(\mathsf{Vararg}\,t\,t'))$

As a direct result, the translation of functions using `vararg` preserves the type translation.

Lemma 5.2.3 *If*

$$|\Delta|; |\Gamma \uplus \{x{:}\tau_1\}| \vdash e' : |\tau_2|,$$

then

$$|\Delta|; |\Gamma| \vdash \texttt{vararg}[|\tau_1|][|\tau_2|](\lambda x{:}T(|\tau_1|).\, e') : |\tau_1 \to \tau_2|.$$

In Chapter 8, I show that for most SML code (and I conjecture code in other similar languages), it is rarely the case that we do not know enough information about the argument type at compile time that we must use `vararg` to choose a calling convention at link- or even run-time. Therefore, most functions will be translated using one of the first two translation rules.

Furthermore, standard optimizations, such as compile-time β-reduction for constructor abstractions, can eliminate variable types at compile time and hence eliminate the need for `vararg`. Indeed, it is entirely reasonable to translate every function using `vararg` and allow an optimizer "fix-up" the inefficiencies. In this fashion, `vararg` reifies the monomorphic term translation of functions in the same way that Vararg reifies the type translation.

Since the source language does not have first-class polymorphism and the scope of a polymorphic value is constrained, we could eliminate all polymorphism at compile time and thus, all occurrences of `vararg`. However, I argued earlier that eliminating all polymorphism at compile time is not reasonable since it duplicates code and does not scale to languages with first-class polymorphism, modules, or separate compilation. I therefore leave the decision to inline a polymorphic function to an optimizer. If polymorphic functions tend to be small or the number of uses is relatively small, then a reasonable strategy is to inline them within their defining compilation unit.

While I borrow the idea of a coercion to mitigate the mismatch in calling conventions, there are still significant differences between my approach and the approach suggested by Leroy (see Section 1.2.3). First, Leroy's coercions can always be calculated at compile-time without the need to examine types at run-time. In contrast, `vararg` may need to examine constructors at run-time. However, I argued that Leroy's approach does not scale to languages like λ_i^{ML} that have first-class polymorphic values, while clearly, my approach can. Second, Leroy's S and G coercions *recursively* pull apart any polymorphic object to coerce its components and make a "deep" copy of a data structure. For instance, if we have a vector of polymorphic functions, Leroy's coercions will traverse and create a new vector when the unknown types are instantiated. In contrast, my "shallow" coercion only affects a closure as it is constructed. In particular, I delay the construction of a vector of polymorphic functions until the unknown types are apparent, at which point the appropriate coercion is selected. Hence, no copy is ever generated and there is no coherency issue when state is involved. Finally, while I use a coercion to mitigate a mismatch with calling conventions, I do not use coercions to implement all language features (see for instance Section 5.4). Thus, `typerec`,Typerec and the ideas of dynamic type dispatch support coercions, but are a much more general mechanism.

5.2.5 Translation of Applications

As with functions, there are three cases to consider when translating an application: the argument type is either a tuple, a non-tuple, or a variable.

If the term is $e_1\, e_2$ and the argument e_2 has a tuple type, I need to extract the components of the tuple and pass them directly to the function. I translate e_1 and e_2, binding the resulting expressions to x_1 and x_2 via `let`. Then, I project the components of x_2 and pass them to x_1.

$$\frac{\Delta;\Gamma \vdash e_1 : \langle \tau_1 \times \tau_2 \rangle \to \tau \Rightarrow e_1' \quad \Delta;\Gamma \vdash e_2 : \langle \tau_1 \times \tau_2 \rangle \Rightarrow e_2'}{\begin{array}{c}\Delta;\Gamma \vdash e_1\, e_2 : \tau \Rightarrow \\ \texttt{let}\, x_1{:}T(|\langle \tau_1 \times \tau_2 \rangle \to \tau|) = e_1' \texttt{ in let}\, x_2{:}T(|\langle \tau_1 \times \tau_2 \rangle|) = e_2' \texttt{ in } x_1\ [\pi_1\, x_2, \pi_2\, x_2]\end{array}}$$

Again, an optimizer should eliminate unnecessary projections. In particular, if e_1' is a tuple $\langle e_a, e_b \rangle$ that is constructed solely to pass multiple arguments, then optimization will yield the simple expression $e_1'\ [e_a, e_b]$.

If the argument has a non-tuple, non-variable type, then the translation is straightforward:

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \tau' \to \tau \Rightarrow e_1' \\ \Delta;\Gamma \vdash e_2 : \tau' \Rightarrow e_2'\end{array}}{\Delta;\Gamma \vdash e_1\, e_2 : \tau \Rightarrow e_1'\, e_2'}$$

If the argument type is a type variable (t), then I must decide whether or not to flatten the argument into multiple arguments using `typecase`. The following `onearg` function calculates a coercion for a function, deciding whether or not to pass the argument flattened, based on t:

$$\begin{array}{lcl}\texttt{onearg} & = & \Lambda t{::}\Omega.t'{::}\Omega. \\ & & \quad \texttt{typecase}\ t\ \mathsf{of} \\ & & \qquad \mathsf{Prod}(t_1, t_2) \texttt{=>} \\ & & \qquad \lambda[f{:}[T(t_1), T(t_2)] \to T(t')].\lambda[x{:}T(\mathsf{Prod}(t_1, t_2))].f\ [\pi_1\, x, \pi_2\, x] \\ & & \quad\ \mid\ \texttt{\_=>}\ \lambda[f{:}[T(t)] \to T(t')].f\end{array}$$

It is easy to verify that `onearg` translates functions from their Vararg calling convention so that they take one argument.

Lemma 5.2.4 $\vdash \texttt{onearg} : \forall t{::}\Omega.\forall t'{::}\Omega.[T(\mathsf{Vararg}\, t\, t')] \to [T(t)] \to T(t')$

Hence, a simple translation of application is as follows:

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \Rightarrow e_1' \quad \Delta;\Gamma \vdash e_2 : \tau_1 \Rightarrow e_2'}{\Delta;\Gamma \vdash e_1\, e_2 : \tau_2 \Rightarrow (\texttt{onearg}[|\tau_1|][|\tau_2|]\, e_1')\ e_2'}$$

The following lemma shows that this translation of application obeys the type translation.

Lemma 5.2.5 *If* $|\Delta|;|\Gamma| \vdash e_1' : |\tau_1 \to \tau_2|$ *and* $|\Delta|;|\Gamma| \vdash e_2' : |\tau_1|$, *then* $|\Delta|;|\Gamma| \vdash (\texttt{onearg}[|\tau_1|][|\tau_2|]\ e_1')\ e_2' : |\tau_2|$.

Again, an optimizer can inline and eliminate the call to `onearg` when $|\tau_1|$ is known and can then decide whether or not to flatten e_2'. Furthermore, the following lemma shows that `onearg` is a left-inverse of `vararg`. This gives an optimizer the opportunity to replace

$$\texttt{onearg}[\mu_1][\mu_2]\,(\texttt{vararg}[\mu_1][\mu_2]\, v)$$

with simply v even when the argument type of the function v is unknown. Henglein and Jörgensen suggest a similar approach to eliminate excessive Leroy-style coercions [65].

Lemma 5.2.6 *If* $\vdash v : T(\mu') \to T(\mu)$, $\vdash v' : T(\mu')$, *then*

$$(\texttt{onearg}[\mu'][\mu](\texttt{vararg}[\mu'][\mu]\ v))\ v' \Downarrow v''$$

iff $v\,v' \Downarrow v''$.

Proof: By induction on the normal form of μ'. □

5.2.6 Translation of Type Abstraction and Application

When translating a `def`, I translate the polymorphic value v yielding v' and bind this using `let`. I translate an instantiation $v[\tau_1, \cdots, \tau_n]$ by applying the translation of v to the constructors generated by $|\tau_1|, \cdots, |\tau_n|$. I translate polymorphic variables to themselves and I translate type-abstractions to constructor abstractions.

This part of the term translation may appear innocuous at first, but it is significant: Traditional compilers ignore type applications and type abstractions since they do not use dynamic type dispatch. Hence, they pass no arguments to polymorphic values when they are instantiated.

In my translation, I turn a type-abstraction into a function that takes constructors as arguments and pass the appropriate constructor arguments to the abstraction at run-time. In some sense, building the constructor arguments and passing them at run-time is the "overhead" of dynamic type dispatch because I must do this whether or not the abstraction examines the constructors via `typerec`. Within a compilation unit, an optimizer may determine that some constructor arguments are not used by a polymorphic function and modify local call sites so that they do not build or pass these constructors at run time. In at least some cases, however, type application will require building and passing constructors at run time.

5.3 Correctness of the Translation

From the lemmas regarding `peq`, `vararg`, and `onearg`, and the commutivity of type translation with substitution, it is easy to show by induction on the derivation of $\Delta; \Gamma \vdash e : \tau \Rightarrow e'$ that the term translation preserves types modulo the type translation.

Lemma 5.3.1 *If* $\Delta; \Gamma \vdash e : \sigma \Rightarrow e'$, *then* $|\Delta|; |\Gamma| \vdash e' : |\sigma|$.

To prove the correctness of the translation, I want to assert that a Mini-ML expression terminates with a value iff its translation terminates with an "equivalent" value. Equivalence of values is easy to define at base types – syntactic equality will do nicely. But what should be the definition of equivalence for arrow types? I need a notion of *semantic* equivalence that captures the idea that functions are equivalent when they compute equivalent answers, given equivalent arguments.

It seems as though I am stuck: To determine whether a Mini-ML *expression* and its translation compute equivalently, I must define what it means for Mini-ML and λ_i^{ML}-Rep *values* to be equivalent. But to define what it means for two values to be equivalent, in particular what it means for two functions to be equivalent, I need to define what it means for two expressions to compute equivalently. How can I formulate these two relations that are defined in terms of one another?

The answer to this dilemma is to simultaneously define these simulation relations, but index the relations by *types*. I will start by defining value equivalence and expression equivalence at closed, base types and then *logically* extend the notions of equivalence for higher types in terms of the equivalence relations indexed by the component types. In the end, I will generate a family of inductively defined relations which will allow us to argue by induction on the *type* of an expression that its translation is correct.

I begin by defining an auxiliary relation between closed, Mini-ML monotypes and closed λ_i^{ML}-Rep constructors that respects constructor equivalence:

$$\frac{\exists \mu'.|\tau| = \mu' \text{ and } \vdash \mu' \equiv \mu :: \Omega}{\tau \approx \mu}$$

In Figure 5.5, I give suitable relations between closed Mini-ML and λ_i^{ML}-Rep terms. The relations are indexed by closed, Mini-ML type schemes. Two computations are related if, whenever one evaluates to a value, then the other evaluates to a related value. Two values are related at base type if they are syntactically equal. Two values are related at a product type if projecting the corresponding components yields related computations at the component type. Two values of arrow type are related if, whenever we have values related at the domain type, applying the functions to the values yields related computations. In the case of the λ_i^{ML}-Rep function, we must first coerce the function to take one argument via the `onearg` function. Finally, values are related at the type scheme $\forall t_1, \cdots, t_n.\tau$, if, whenever applied to closed, related monotypes and constructors, they yield related computations at the type obtained by substituting the monotypes for the type variables.

The relations $e \sim_\tau e'$, $v \approx_\tau v'$, and $v \approx_\sigma v'$ are well founded even though their definitions refer to one another, because either the size of the type index decreases, or else the number of quantifiers in the type index decreases. This is ensured because Mini-ML is *predicative*, (i.e., only monotypes can instantiate type variables).

The monotype/constructor relation is extended to substitutions δ and δ', indexed by a set of type variables, Δ, where δ maps type variables to closed, Mini-ML monotypes

Expression Relations:

$$\frac{e \Downarrow v \text{ iff } e' \Downarrow v' \text{ and } v \approx_\tau v'}{e \sim_\tau e'}$$

Value Relations:

$$i \approx_{\mathsf{int}} i \qquad f \approx_{\mathsf{float}} f \qquad \langle\rangle \approx_{\mathsf{unit}} \langle\rangle$$

$$\frac{\pi_1\, v \sim_{\tau_1} \pi_1\, v' \qquad \pi_2\, v \sim_{\tau_2} \pi_2\, v'}{v \approx_{\langle \tau_1 \times \tau_2 \rangle} v'}$$

$$\frac{\forall v_1, v_1'.v_1 \approx_{\tau'} v_1' \text{ implies } v\, v_1 \sim_\tau (\texttt{onearg}[|\tau'|][|\tau|]\, v')\, (v_1')}{v \approx_{\tau' \to \tau} v'}$$

$$\frac{\begin{array}{c}\forall \tau_1, \mu_1, \cdots, \tau_n, \mu_n.\tau_1 \approx \mu_1, \cdots, \tau_n \approx \mu_n \text{ implies} \\ v[\tau_1, \cdots, \tau_n] \sim_{\{\tau_1/t_1, \cdots, \tau_n/t_n\}\tau} v'[\mu_1] \cdots [\mu_n]\end{array}}{v \approx_{\forall t_1, \cdots, t_n.\tau} v'}$$

Figure 5.5: Relating Mini-ML to λ_i^{ML}-Rep

and δ' maps type variables to closed, λ_i^{ML}-Rep constructors as follows:

$$\frac{\begin{array}{c} Dom(\delta) = Dom(\delta') = Dom(\Delta) \\ \forall t \in \Delta.\delta(t) \approx \delta'(t)\end{array}}{\delta \approx_\Delta \delta'}$$

The term relation is extended to pairs of substitutions, $\delta; \gamma$ and $\delta'; \gamma'$, indexed by $\Delta; \Gamma$, where δ and δ' are as above and γ and γ' are substitutions from term variables to values. I assume that all free type variables occurring in the range of Γ are in Δ.

$$\frac{\begin{array}{c}\delta \approx_\Delta \delta' \qquad \Delta \vdash \Gamma \qquad Dom(\gamma) = Dom(\gamma') = Dom(\Gamma) \\ \forall x \in Dom(\Gamma).\delta(\gamma(x)) \approx_{\Gamma(x)} \gamma'(x)\end{array}}{\delta; \gamma \approx_{\Delta;\Gamma} \delta'; \gamma}$$

With these definitions, I can begin to establish the correctness of the term translation. The first step is to show that `peq` has the appropriate behavior.

Lemma 5.3.2 *If* $v_1 \approx_\tau v_1'$ *and* $v_2 \approx_\tau v_2'$, *then* $\texttt{eq}(v_1, v_2) \sim_{\mathsf{int}} \texttt{peq}[|\tau|][v_1', v_2']$.

Proof: By induction on τ. □

Next, I argue that, under appropriate circumstances, abstracting related values with respect to related expressions yields related functions. This follows almost directly from the definitions of the relations and the fact that `onearg` and `vararg` are left-inverses.

Lemma 5.3.3 *Suppose* $\Delta; \Gamma \uplus \{x{:}\tau'\} \vdash e : \tau \Rightarrow e'$, *and for all* $\delta; \gamma \uplus \{x{=}v\} \approx_{\Delta;\Gamma\uplus\{x:\tau'\}} \delta'; \gamma' \uplus \{x{=}v'\}$, $\delta(\gamma \uplus \{x{=}v\}(e)) \sim_{\delta(\tau)} \delta'(\gamma' \uplus \{x{=}v'\}(e'))$. *Then for all* $\delta; \gamma \approx_{\Delta;\Gamma} \delta'; \gamma'$, $\delta(\gamma(\lambda x{:}\tau'.e)) \approx_{\delta(\tau'\to\tau)} \delta'(\gamma'(\mathtt{vararg}[|\tau'|][|\tau|](\lambda x{:}T(|\tau'|).e')))$

Proof: Let $\delta; \gamma \approx_{\Delta;\Gamma} \delta'; \gamma'$ and let $v \approx_{\delta(\tau')} v'$. I must show:

$(\delta(\gamma(\lambda x{:}\tau'.e)))\ v \sim_{\delta(\tau)} (\mathtt{onearg}[\delta'(|\tau'|)][\delta'(|\tau|)](\delta'(\gamma'(\mathtt{vararg}[|\tau'|][|\tau|](\lambda x{:}T(|\tau'|).e')))))\ v'$.

This holds iff:

$$\begin{aligned}(\lambda x{:}\delta(\tau').\delta(\gamma(e)))\ v \quad \sim_{\delta(\tau)} \quad & (\mathtt{onearg}[\delta'(|\tau'|)][\delta'(|\tau|)] \\ & \quad (\mathtt{vararg}[\delta'(|\tau'|)][\delta'(|\tau|)](\lambda x{:}T(\delta'(|\tau'|)).\delta'(\gamma'(e')))))\ v'.\end{aligned}$$

Since `onearg` is a left-inverse of `vararg` (see lemma 5.2.6), this holds iff:

$$(\lambda x{:}\delta(\tau').\delta(\gamma(e)))\ v \sim_{\delta(\tau)} (\lambda x{:}T(\delta'(|\tau'|)).\delta'(\gamma'(e')))\ v'$$

which holds iff:

$$\delta(\gamma \uplus \{x{=}v\}(e)) \sim_{\delta(\tau)} \delta'(\gamma' \uplus \{x{=}v'\}(e'))$$

which follows by assumption. □

Finally, I establish the correctness of the translation by showing that applying related substitutions to an expression and its translation yields related computations.

Theorem 5.3.4 *If* $\Delta; \Gamma \vdash e : \sigma \Rightarrow e'$ *and* $\delta; \gamma \approx_{\Delta;\Gamma} \delta'; \gamma'$, *then* $\delta(\gamma(e)) \sim_{\delta(\sigma)} \delta'(\gamma'(e'))$.

Proof: By induction on the derivation of $\Delta; \Gamma \vdash e : \tau \Rightarrow e'$. Let $\delta; \gamma \approx_{\Delta;\Gamma} \delta'; \gamma'$. The **int**, **float**, and **unit** cases follow trivially. The **var** case follows from the assumptions regarding $\delta; \gamma$ and $\delta'; \gamma'$. The equality case follows from lemma 5.3.2. The **if0** case follows since related values at `int` must be the same integer. The **pair** case follows from the inductive assumptions and the **proj** case follows from the definition of the relations at product types. The **abs** case follows from lemma 5.3.3 and the **app** case follows from the definition of the relations at arrow types. The **tapp** case follows from the definition of the relations at type schemes, and the fact that $\delta(\tau) \approx \delta'(|\tau_i|)$ by lemma 5.2.1. □

Corollary 5.3.5 (Translation Correctness) *If* $\vdash e : \sigma \Rightarrow e'$, *then* $e \sim_\sigma e'$.

Proof: Suppose $\emptyset; \emptyset \vdash e : \sigma \Rightarrow e'$. Taking $\delta = \delta' = \emptyset$ and $\gamma = \gamma' = \emptyset$, we have $\delta; \gamma \approx_{\emptyset;\emptyset} \delta'; \gamma'$. By the previous theorem, then $\delta(\gamma(e)) \sim_{\delta(\sigma)} \delta'(\gamma'(e'))$, thus $e \sim_\sigma e'$.
□

To summarize, I have defined a translation from Mini-ML to λ_i^{ML}-Rep that eliminates polymorphic equality and flattens function arguments. The type translation uses Typerec to define Vararg, which determines calling conventions for functions. The term translation uses `typerec` to define `peq`, `vararg`, and `onearg`. The `vararg` term converts a function from taking one argument so that it has the proper calling convention according to Vararg. Conversely, the `onearg` term converts a function from its Vararg calling convention to one argument.

A real source language like SML provides n-tuples (for arbitrary n) instead of only binary tuples. Extending the flattening translation so that it flattens argument tuples of less than k components is straightforward – we simply use $k+1$ cases in the definitions of Vararg to determine the proper calling convention:

$$
\begin{array}{rcl}
\mathsf{Vararg} & = & \lambda t{::}\Omega.\lambda t'{::}\Omega. \\
& & \quad \mathsf{Typecase}\ t\ \mathsf{of} \\
& & \qquad \mathsf{Prod}(t_1) \Rightarrow \mathsf{Arrow}(t_1, t') \\
& & \qquad \mid \mathsf{Prod}(t_1, t_2) \Rightarrow \mathsf{Arrow}([t_1, t_2], t') \\
& & \qquad \mid \mathsf{Prod}(t_1, t_2, t_3) \Rightarrow \mathsf{Arrow}([t_1, t_2, t_3], t') \\
& & \qquad\quad \cdots \\
& & \qquad \mid \mathsf{Prod}(t_1, \cdots, t_k) \Rightarrow \mathsf{Arrow}([t_1, \cdots, t_k], t') \\
& & \qquad \mid \_ \Rightarrow \mathsf{Arrow}(t, t')
\end{array}
$$

Similarly, the definitions of `vararg` and `onearg` will require $k+1$ cases.

5.4 Compiling Other Constructs

Dynamic type dispatch can be used to support a variety of language mechanisms in the presence of unknown types. In this section, I show how the dynamic type dispatch facilities of λ_i^{ML} can be used to support flattened data structures (such as C-style structs and arrays), Haskell-style type classes [53], and polymorphic communication primitives.

5.4.1 C-style Structs

Languages like C provide flattened data structures by default. Programmers explicitly specify when they want to use pointers. This gives programmers control over both sharing and data layout. For example, a C struct (i.e., record) with nested struct components such as

```
struct {
   struct {int x; double y;} a;
   int b;
   struct {double f; double g;} c;
},
```

is typically represented in the same way as a flattened struct made out of the primitive components (ignoring alignment constraints):

```
struct {
   int x;
   double y;
   int b;
   double f;
   double g;
}.
```

In effect, a C compiler performs a type-directed translation that eliminates nested structs. To perform this flattening, a C compiler relies upon there being no unknown types at compile time. In this section, I show how to use dynamic type analysis to flatten structs in the presence of unknown types.

I begin by extending λ_i^{ML}-Rep to support records with an arbitrary number of primitive components. To this end, I add list kinds (κ^*) with introductory constructors Nil_κ and $\mathsf{Cons}(\mu_1, \mu_2)$, and an eliminatory constructor $\mathsf{Listrec}\,\mu\,\mathsf{of}\,(\mu_n; \mu_c)$. Similar to Typerec, the Listrec constructor provides a means for folding a computation across a list of constructors. I then replace Unit and $\mathsf{Prod}(\mu_1, \mu_2)$ with a single constructor $\mathsf{Struct}(\mu)$, where μ is a constructor of kind Ω^* (i.e., a list of monotypes). As before, we can generate the monotypes by induction; but for Structs, we require a dual induction to generate lists of monotypes. Therefore, I extend the Typerec constructor to provide a means for folding a computation across the list of Ω components of a Struct constructor. The resulting grammars for kinds and constructors are as follows:

$$
\begin{array}{llcl}
\text{(kinds)} & \kappa & ::= & \cdots \mid \kappa^* \\[1em]
\text{(constructors)} & \mu & ::= & \cdots \mid \mathsf{Nil}_\kappa \mid \mathsf{Cons}(\mu_1, \mu_2) \mid \mathsf{Struct}(\mu) \mid \\
& & & \mathsf{Listrec}\,\mu\,\mathsf{of}\,(\mu_n; \mu_c) \mid \\
& & & \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_s; \mu_a)(\mu_n; \mu_c)
\end{array}
$$

The formation rules for the constructors are straightforward except for Listrec and Typerec. A Listrec is well-formed with kind κ, provided its argument is a list, and it maps an empty list to a constructor of kind κ_1 and a non-empty list to a constructor of kind κ_1, given the head and tail of the list as arguments, as well as the result of folding the Listcase across the tail of the list.

$$
\frac{\Delta \vdash \mu_n :: \kappa \qquad \begin{array}{c}\Delta \vdash \mu :: {\kappa_1}^* \\ \Delta \vdash \mu_c :: \kappa_1 \to {\kappa_1}^* \to \kappa \to \kappa\end{array}}{\Delta \vdash \mathsf{Listrec}\,\mu\,\mathsf{of}\,(\mu_n; \mu_c) :: \kappa}
$$

The formation rule for $\mathsf{Typerec}$ is as before, but we require extra constructors μ_n and μ_c in order to fold the $\mathsf{Typerec}$ across the list components of a Struct. In effect, we simultaneously define a $\mathsf{Typerec}$ with a $\mathsf{Listrec}$. Therefore, the formation rule for $\mathsf{Typerec}$ is as follows[1]:

$$\frac{\begin{array}{c}\Delta \vdash \mu :: \Omega \quad \Delta \vdash \mu_i :: \kappa \quad \Delta \vdash \mu_f :: \kappa \\ \Delta \vdash \mu_a :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \quad \Delta \vdash \mu_s :: \kappa' \to \kappa \\ \Delta \vdash \mu_n \quad \Delta \vdash \mu_c :: \Omega \to \Omega^* \to \kappa \to \kappa' \to \kappa'\end{array}}{\Delta \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a; \mu_s)(\mu_n; \mu_c) :: \kappa}$$

The μ_n and μ_c clauses determine how the $\mathsf{Typerec}$ computation is folded across the components of a Struct, resulting in a constructor of kind κ'. The μ_s clause simply converts this κ' constructor to a constructor of kind κ.

The equivalences that govern $\mathsf{Listrec}$ and $\mathsf{Typerec}$ are as before: We choose the appropriate clause according to the head of the normal form of the argument, and unroll the computation on the component constructors. For a Struct, we unroll as follows:

$$\begin{array}{l}\mathsf{Typerec}\ \mathsf{Struct}(\mu)\ \mathsf{of}\ (\mu_i; \mu_f; \mu_a; \mu_s)(\mu_n; \mu_c) \equiv \\ \quad \mu_s\,(\mathsf{Listrec}\,\mu\,\mathsf{of}\,(\mu_n; (\lambda t_a{::}\Omega.\lambda t_b{::}\Omega^*.\lambda t_b'{::}\kappa'. \\ \qquad\qquad\qquad \mu_c\, t_a\, t_b\,(\mathsf{Typerec}\ t_a\ \mathsf{of}\ (\mu_i; \mu_f; \mu_a; \mu_s)(\mu_n; \mu_c))\, t_b')))\end{array}$$

To the types, I add $\mathsf{struct}\{\sigma_1, \cdots, \sigma_n\}$ for $n \geq 0$, as well as the following equivalence relating Struct constructors and struct types:

$$\frac{\Delta \vdash \mu_1 :: \Omega \quad \cdots \quad \Delta \vdash \mu_n :: \Omega}{\Delta \vdash T(\mathsf{Struct}(\mathsf{Cons}(\mu_1, \cdots \mathsf{Cons}(\mu_n, \mathsf{Nil}_\Omega)))) \equiv \mathsf{struct}\{T(\mu_1), \cdots, T(\mu_n)\}}$$

To the terms, I first add `listrec` so that we may fold a term-level computation across a list of constructors:

$$(\mathbf{lrec})\ \frac{\begin{array}{c}\Delta \vdash \mu :: \kappa^* \quad \Delta; \Gamma \vdash e_n : \{\mathsf{Nil}_\kappa/t\}\sigma \\ \Delta; \Gamma \vdash e_c : \forall t_1{::}\kappa.\forall t_2{::}\kappa^*.\{t_2/t\}\sigma \to \{\mathsf{Cons}(t_1, t_2)/t\}\sigma\end{array}}{\Delta; \Gamma \vdash \texttt{listrec}\ \mu\ \texttt{of}\ [t{::}\kappa^*.\sigma](e_n; e_c) : \{\mu/t\}\sigma}$$

As I did at the constructor-level, I extend `typerec` so that we simultaneously define how to fold a term-level computation across types and across lists of type:

$$(\mathbf{trec})\ \frac{\begin{array}{c}\Delta \vdash \mu :: \Omega \quad \Delta; \Gamma \vdash e_i : \{\mathsf{Int}/t\}\sigma \quad \Delta; \Gamma \vdash e_f : \{\mathsf{Float}/t\}\sigma \\ \Delta; \Gamma \vdash e_a : \forall t_1{::}\Omega.\forall t_2{::}\Omega.\{t_1/t\}\sigma \to \{t_2/t\}\sigma \to \{\mathsf{Arrow}(t_1, t_2)/t\}\sigma \\ \Delta; \Gamma \vdash e_s : \forall t_1{::}\Omega^*.\{t_1/t'\}\sigma' \to \{\mathsf{Struct}(t_1)/t\}\sigma\end{array}}{\Delta; \Gamma \vdash \texttt{typerec}\ \mu\ \texttt{of}\ [t{::}\Omega.\sigma](e_i; e_f; e_a; e_s)[t'{::}\Omega^*.\sigma'](e_n; e_c) : \{\mu/t\}\sigma}$$

[1] To simplify the presentation, I only consider single argument functions at the term level, and hence Arrow takes one domain constructor instead of k. It is straightforward to extend Arrow to take and return an arbitrary number of arguments and results by giving it kind $\Omega^* \to \Omega^* \to \Omega$.

I also add two sorts of intro and elim forms for structs. The first sort provides both an efficient mechanism for constructing structs ($\texttt{struct}\{e_1, \cdots, e_n\}$) and an efficient mechanism for projecting a component from a struct ($\texttt{\#i}\, e$). The typing rules for these terms are standard:

$$(\textbf{struct})\ \frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \quad \cdots \quad \Delta;\Gamma \vdash e_n : \sigma_n}{\Delta;\Gamma \vdash \texttt{struct}\{e_1, \cdots, e_n\} : \mathsf{struct}\{\sigma_1, \cdots, \sigma_n\}} \quad (n \geq 0)$$

$$(\textbf{select})\ \frac{\Delta;\Gamma \vdash e : \mathsf{struct}\{\sigma_1, \cdots, \sigma_n\}}{\Delta;\Gamma \vdash \texttt{\#i}\, e : \sigma_i} \quad (1 \leq i \leq n)$$

However, these terms do not provide a means for constructing or deconstructing a struct by induction on the list of component types. Consider, for example, extending the polymorphic equality term of the previous section to compare arbitrary structs:

$$\begin{array}{lcl} \mathsf{peq}[\mathsf{Int}] & = & \lambda[x_1{:}\mathsf{int}, x_2{:}\mathsf{int}].\, \texttt{eqint}(x_1, x_2) \\ \mathsf{peq}[\mathsf{Float}] & = & \lambda[x_1{:}\mathsf{float}, x_2{:}\mathsf{float}].\, \texttt{eqfloat}(x_1, x_2) \\ \mathsf{peq}[\mathsf{Struct}(t)] & = & \lambda[x_1{:}T(\mathsf{Struct}(t)), x_2{:}T(\mathsf{Struct}(t))].\, ??? \end{array}$$

I need to project the components of the structure and compare them at their respective types. I cannot use **select** since both $\texttt{i}$ and n must be determined at compile time, and the length of the list of constructors t is unknown. I therefore add a second sort of intro and elim forms that allows us to construct ($\texttt{cons}(e_1, e_2)$) and deconstruct structs ($\texttt{head}\, e$ and $\texttt{tail}\, e$) by induction on the list of components. These terms have the following formation rules:

$$(\textbf{cons})\ \frac{\Delta;\Gamma \vdash e_1 : T(\mu_1) \quad \Delta;\Gamma \vdash e_2 : T(\mathsf{Struct}(\mu_2))}{\Delta;\Gamma \vdash \texttt{cons}(e_1, e_2) : T(\mathsf{Struct}(\mathsf{Cons}(\mu_1, \mu_2)))}$$

$$(\textbf{hd})\ \frac{\Delta;\Gamma \vdash e : T(\mathsf{Struct}(\mathsf{Cons}(\mu_1, \mu_2)))}{\Delta;\Gamma \vdash \texttt{head}\, e : T(\mu_1)}$$

$$(\textbf{tl})\ \frac{\Delta;\Gamma \vdash e : T(\mathsf{Struct}(\mathsf{Cons}(\mu_1, \mu_2)))}{\Delta;\Gamma \vdash \texttt{tail}\, e : T(\mathsf{Struct}(\mu_2))}$$

Operationally, `cons` takes values v and $\texttt{struct}\{v_1, \cdots, v_n\}$, and constructs the new value $\texttt{struct}\{v, v_1, \cdots, v_n\}$. Correspondingly, `head` and `tail` take a value $\texttt{struct}\{v_1, v_2, \cdots, v_n\}$ and return values v_1 and $\texttt{struct}\{v_2, \cdots, v_n\}$, respectively.

It is possible to effectively define the other struct primitives with `cons`, `head`, and `tail`[2]. For example, $\texttt{struct}\{e_1, e_2, \cdots, e_n\}$ can be defined as

$$\texttt{struct}\{e_1, e_2, \cdots, e_n\} \quad = \quad \texttt{cons}(e_1, \texttt{cons}(e_2, \cdots \texttt{cons}(e_n, \texttt{struct}\{\}) \cdots)),$$

[2]The encoding is not quite complete, because `cons`, `head`, and `tail` only operate on structs of monotypes, whereas the other operations can operate on structs of polytypes.

whereas `#i` e can be defined as

$$\begin{array}{rcl} \texttt{\#1}\,e & = & \texttt{head}\,e \\ \texttt{\#i}\,e & = & \texttt{\#(i-1)}\,(\texttt{tail}\,e) \end{array}$$

However, these encodings generate many intermediate structs that are simply discarded. Some implementation strategies may avoid creating most if not all of these structs. For instance, in C, the `tail` operation can be implemented by returning the address of the second component of a struct. Unfortunately, many memory management strategies forbid pointers into the middle of objects. TIL implements `tail` by returning a logical pointer or *cursor* instead of an actual pointer (see Chapter 8). The logical pointer is implemented as a pair, which consists of a pointer to the beginning of the original struct and an integer offset. Adding the offset to the pointer yields the logical pointer. This approach is compatible with most memory management strategies and provides an $O(1)$ `tail`. Unfortunately, this still makes projecting the i^{th} component an $O(i)$ operation. Therefore, I use the select operation (`#i` e) for $O(1)$ access whenever possible, and only use `head` and `tail` when iterating across a struct. Likewise, I use $\texttt{struct}\{e_1, \cdots, e_n\}$ whenever possible to avoid creating any intermediate structs, and only use `cons` when necessary.

With these additions to the target language, I can now define a translation that maps Mini-ML tuples to flattened structs. Of course, flattening all tuples is not a good strategy, but the source language can provide two sorts of tuple types — those that should be flattened and those that should not — and hence leave the choice of representation to the programmer (as in C). Alternatively, a compiler may perform some analysis to determine a representation that is likely to be beneficial. To demonstrate the key ideas, I will simply assume that all products are to be flattened.

The type translation is as before (see Section 5.2.1), except for unit and products. The translation of unit is simply an empty structure. In the translation of a tuple type, I use two auxiliary constructor functions, Flat and Append:

$$\begin{array}{rcl} |\mathsf{unit}| & = & \mathsf{Struct}(\mathsf{Nil}_\Omega) \\ |\langle\tau_1 \times \tau_2\rangle| & = & \mathsf{Struct}(\mathsf{Append}[\mathsf{Flat}[|\tau_1|]][\mathsf{Flat}[|\tau_2|]]) \end{array}$$

These two functions are defined using the pattern matching notation for Typecase and Listrec as follows: Flat takes a monotype and, if it is a Struct, returns the list of components of the Struct. Otherwise, Flat conses the given monotype onto nil to create a singleton list:

$$\mathsf{Flat} :: \Omega \to \Omega^*$$

$$\begin{array}{rcll} \mathsf{Flat} & = & \lambda t{::}\Omega.\mathsf{Typecase}\ t\ \mathsf{of} & \mathsf{Struct}(t) \mathrel{=>} t \\ & & & \mid \_ \mathrel{=>} \mathsf{Cons}(t, \mathsf{Nil}_\Omega)) \end{array}$$

Append takes two lists of monotypes and returns their concatenation:

$$\mathsf{Append} :: \Omega^* \to \Omega^* \to \Omega^*$$

$$\begin{array}{lcl} \mathsf{Append}[\mathsf{Nil}_\Omega][t] & = & t \\ \mathsf{Append}[\mathsf{Cons}(t_1, t_2)][t] & = & \mathsf{Cons}(t_1, \mathsf{Append}[t_2][t]) \end{array}$$

Therefore, the type translation of a product results in a struct where the structs resulting from nested product components have been flattened and appended together.

The term translation is as before except for tuple creation and projection. Unit simply maps to an empty struct. Binary tuples are translated according to the following rule:

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \Rightarrow e_1' \quad \Delta; \Gamma \vdash e_2 : \tau_1 \Rightarrow e_2'}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \langle \tau_1 \times \tau_2 \rangle \Rightarrow \mathsf{append}[\mathsf{Flat}|\tau_1|][\mathsf{Flat}|\tau_2|](\mathtt{flat}[\tau_1]\, e_1)\, (\mathtt{flat}[\tau_2]\, e_2)}$$

The translation uses auxiliary term functions `flat` and append, which are defined using `typecase` and `listrec` as follows: The `flat` function takes a constructor t and a value of type $T(t)$. If the value is a structure, it simply returns that value. If the value is not a structure, then `flat` places it in a structure.

$$\mathtt{flat} : \forall t{::}\Omega. T(t) \to T(\mathsf{Flat}[t])$$

$$\begin{array}{lcl} \mathtt{flat} & = & \Lambda t{::}\Omega.\mathtt{typecase}\ t\ \mathtt{of} \\ & & \quad\quad \mathsf{Struct}(t) \texttt{=>} \lambda x{:}T(\mathsf{Struct}(t)).x \\ & & \quad\ \mid \_ \texttt{=>} \lambda x{:}T(t).\mathtt{struct}\{x\} \end{array}$$

The append function takes two structs and concatenates their contents, yielding a flattened struct:

$$\mathsf{append} : \forall t_1{::}\Omega^*.\forall t_2{::}\Omega^*.T(\mathsf{Struct}(t_1)) \to T(\mathsf{Struct}(t_2)) \to T(\mathsf{Struct}(\mathsf{Append}[t_1][t_2]))$$

$$\begin{array}{lcl} \mathsf{append}[\mathsf{Nil}_\Omega][t] & = & \lambda x{:}\mathsf{struct}\{\}.\lambda y{:}T(\mathsf{Struct}(t)).y \\ \mathsf{append}[\mathsf{Cons}(t_1, t_2)][t] & = & \lambda x{:}T(\mathsf{Struct}(\mathsf{Cons}(t_1, t_2))).\lambda y{:}T(\mathsf{Struct}(t)). \\ & & \quad \mathtt{cons}(\mathtt{head}\, x, \mathsf{append}[t_2][t]\, (\mathtt{tail}\, x)\, y) \end{array}$$

We can easily verify from the types of `flat` and append that the term translation for products respects the type translation.

The term translation of first and second tuple projections is given by the following two inference rules:

$$\frac{\Delta; \Gamma \vdash e : \langle \tau_1 \times \tau_2 \rangle \Rightarrow e'}{\Delta; \Gamma \vdash \pi_1\, e : \tau_1 \Rightarrow \mathtt{unflat}\, |\tau_1| (\mathtt{proj1}[\mathsf{Flat}|\tau_1|]\, [\mathsf{Flat}|\tau_2|]\, e')}$$

$$\frac{\Delta; \Gamma \vdash e : \langle \tau_1 \times \tau_2 \rangle \Rightarrow e'}{\Delta; \Gamma \vdash \pi_2\, e : \tau_1 \Rightarrow \texttt{unflat}\, |\tau_2| (\texttt{proj2}[\mathsf{Flat}|\tau_1|]\, [\mathsf{Flat}|\tau_2|]\, e')}$$

These translations use the auxiliary term functions `unflat`, `proj1` and `proj2`, which are defined as follows: The `unflat` function is the inverse of `flat`. It has type

$$\texttt{unflat} : \forall t{::}\Omega. T(\mathsf{Flat}[t]) \rightarrow T(t),$$

and is defined as:

$$\begin{array}{lcl} \texttt{unflat} & = & \Lambda t{::}\Omega.\texttt{typecase}\ t\ \texttt{of} \\ & & \qquad\qquad \mathsf{Struct}(t) \texttt{=>} \lambda x{:}T(\mathsf{Struct}(t)).x \\ & & \qquad\quad\ \mid \_ \texttt{=>} \lambda x{:}T(\mathsf{Struct}(t)).\texttt{head}\, x \end{array}$$

The `proj1` function extracts the first components of a struct, corresponding to its first argument list of constructors. Similar to append, `proj1` is defined using the term-level `listrec`:

$$\texttt{proj1} : \forall t_1{::}\Omega^*.\forall t_2{::}\Omega^*.T(\mathsf{Struct}(\mathsf{Append}[t_1][t_2])) \rightarrow T(\mathsf{Struct}(t_1))$$

$$\begin{array}{lcl} \texttt{proj1}[\mathsf{Nil}_\Omega][t] & = & \lambda x{:}T(\mathsf{Struct}(t)).\texttt{struct}\{\} \\ \texttt{proj1}[\mathsf{Cons}(t_1, t_2)][t] & = & \lambda x{:}T(\mathsf{Struct}(\mathsf{Append}[\mathsf{Cons}(t_1, t_2)][t])). \\ & & \qquad \texttt{cons}(\texttt{head}\, x, \texttt{proj1}[t_2][t](\texttt{tail}\, x)) \end{array}$$

The `proj2` function extracts the latter components of a struct, corresponding to its second argument list of constructors.

$$\texttt{proj2} : \forall t_1{::}\Omega^*.\forall t_2{::}\Omega^*.T(\mathsf{Struct}(\mathsf{Append}[t_1][t_2])) \rightarrow T(\mathsf{Struct}(t_2))$$

$$\begin{array}{lcl} \texttt{proj2}[\mathsf{Nil}_\Omega][t] & = & \lambda x{:}T(\mathsf{Struct}(t)).x \\ \texttt{proj2}[\mathsf{Cons}(t_1, t_2)][t] & = & \lambda x{:}T(\mathsf{Struct}(\mathsf{Append}[\mathsf{Cons}(t_1, t_2)][t])). \\ & & \qquad \texttt{proj2}[t_2][t](\texttt{tail}\, x) \end{array}$$

The crucial step in showing that the term translations of projections respect the type translation, is showing that

$$\mathsf{Append}[\mathsf{Cons}(\mu_1, \mu_2)][\mu] \equiv \mathsf{Cons}(\mu_1, \mathsf{Append}[\mu_2][\mu]),$$

which follows directly from the definition of Append. This allows us to argue that the inductive cases are well-formed.

One advantage of explicitly flattening structs in the target language is that we can export a type-safe form of casting to the source level. I call such a case a *view*. Let us define two Mini-ML types τ_1 and τ_2 to be similar, $\tau_1 \sim \tau_2$, iff they have the same representations — that is, iff $|\tau_1|$ is definitionally equivalent to $|\tau_2|$. If $\tau_1 \sim \tau_2$ in the source language, then it is possible to safely view any source τ_1 expression as having type τ_2 and vice versa. In particular, given the flattening translation above, any two source tuple types that are equivalent modulo associativity of the tuple constructor, have translations that are definitionally equivalent. Thus, if $v : \tau_1 \times (\tau_2 \times \tau_3)$, we can safely view v as having type $(\tau_1 \times \tau_2) \times \tau_3$.

Because I represent equivalent source types using equivalent target types, no coercion needs to take place when viewing a value with a different, but similar type. Hence, this approach to views, unlike coercion-based approaches, is compatible with mutable types (i.e., arrays and refs) in the sense that $\mathsf{array}[\tau_1] \sim \mathsf{array}[\tau_2]$ whenever $\tau_1 \sim \tau_2$. This means we may freely intermingle updates with views of complex data structures, capturing some of the expressiveness of C casts without sacrificing type-safety.

It is possible to define more sophisticated translations that, for instance, insert padding to ensure that each element of a struct lies on a multiple-of-eight (i.e., quad-word) boundary, assuming the struct is allocated on an aligned boundary. For example, we can modify Append to insert padding (a pointer to an empty struct) between non-Float components:

$$\begin{array}{lcl}
\mathsf{Append}'[\mathsf{Nil}_\Omega][\mu] & = & \mu \\
\mathsf{Append}'[\mathsf{Cons}(\mathsf{Float}, \mu_2)][\mu] & = & \mathsf{Cons}(\mathsf{Float}, \mathsf{Append}'[\mu_2][\mu]) \\
\mathsf{Append}'[\mathsf{Cons}(t_1, t_2)][\mu] & = & \mathsf{Cons}(t_1, \mathsf{Cons}(\mathsf{Struct}(\mathsf{Nil}_\Omega), \mathsf{Append}'[\mu_2][\mu]))
\end{array}$$

Alternatively, we might split the float and non-float components of a struct to avoid padding altogether. This yields the following alternative type translation:

$$|\langle \tau_1 \times \tau_2 \rangle| \quad = \quad \mathsf{Struct}(\mathsf{Append}[\mathsf{Split}[\mathsf{Flat}[|\tau_1|]][\mathsf{Split}[\mathsf{Flat}[|\tau_2|]]]])$$

where Split is defined as

$$\begin{array}{lcl}
\mathsf{Split}[t] & = & \mathsf{Split}'[t][\mathsf{Nil}_\Omega][\mathsf{Nil}_\Omega] \\
\\
\mathsf{Split}'[\mathsf{Nil}_\Omega][t_f][t] & = & \mathsf{Append}[\mathsf{Rev}[t_f]][\mathsf{Rev}[t]] \\
\mathsf{Split}'[\mathsf{Cons}(\mathsf{Float}, t_2)][t_f][t] & = & \mathsf{Split}'[t_2][\mathsf{Cons}(\mathsf{Float}, t_f)][t] \\
\mathsf{Split}'[\mathsf{Cons}(t_1, t_2)][t_f][t] & = & \mathsf{Split}'[t_2][t_f][\mathsf{Cons}(t_1, t)] \\
\\
\mathsf{Rev}[t] & = & \mathsf{Rev}'[t][\mathsf{Nil}_\Omega] \\
\\
\mathsf{Rev}'[\mathsf{Nil}_\Omega][t] & = & t \\
\mathsf{Rev}'[\mathsf{Cons}(t_1, t_2)][t] & = & \mathsf{Rev}'[t_2][\mathsf{Cons}(t_1, t)]
\end{array}$$

This translation maps the Mini-ML type

$$\mathsf{int} \times (\mathsf{float} \times (\mathsf{int} \times (\mathsf{float} \times \mathsf{int})))$$

to the target type $\mathsf{struct}\{\mathsf{float}, \mathsf{float}, \mathsf{int}, \mathsf{int}, \mathsf{int}\}$. Assuming that such values are allocated on quad-word boundaries, the floating-point components will always be aligned.

There is, of course, a limit to the transformations that can be coded, since definitional equivalence of constructors must be decidable so that in turn, type-checking remains decidable. Nevertheless, the range of transformations that λ_i^{ML}-like languages can support seems to cover a wide variety of the interesting cases.

5.4.2 Type Classes

The programming language Haskell [68] gives the programmer the ability to define a *class* of types with associated operations called methods. The canonical example is the class of types that admit equality (also known as equality types in SML [90]). The class of equality types includes primitive types, such as int and float, that have a primitive notion of equality. Equality types also include data structures, such as tuples, when the component types are equality types. However, equality types exclude arrow types since determining whether two functions are extensionally equivalent is generally undecidable.

The peq operation of Section 5.2.3 effectively defines an equality method for equality types. However, the definition includes a case for arrow types, because the type of peq is:

$$\forall t{::}\Omega.[T(t), T(t)] \to \mathsf{int}$$

and t ranges over all monotypes, not just the class of equality types. We would like to restrict peq so that only equality types can be passed to it.

SML accomplishes such a restriction for its polymorphic equality operation by having two classes of type variables: Normal type variables (e.g., α) may be instantiated with any monotype, and equality type variables (e.g., $''\alpha$) may only be instantiated with equality types. The polymorphic equality operation is assigned the SML type:

$$\forall ''\alpha.''\alpha \times ''\alpha \to \mathsf{bool}$$

Hence, only equality types may instantiate the polymorphic equality primitive. In particular, an SML type checker will reject the following expression:

fn (x:$\alpha \to \beta$,y:$\alpha \to \beta$) => x = y.

Haskell generalizes this sort of restriction by qualifying bound type variables with a user-defined predicate or predicates (e.g., is_eqty(α))[3]. Another approach, suggested by Duggan [38], is to *refine* the kind of the bound type variable, much as Freeman and Pfenning suggest refinements of SML datatypes [44].

[3]See Jones [72, 71] for a general formulation of qualified types.

However, it is possible to encode type classes to a limited degree using Typerec. In this section, I demonstrate the encoding by sketching how the equality types of SML can be simulated in λ_i^{ML}. The basic idea is to represent the type class as a constructor function that maps equality types to themselves, and non-equality types to the distinguished constructor Void. This Void constructor corresponds to the void type, which is empty. That is, there are no closed values of type void.

As an example, the class of equality types is encoded by the following constructor function Eq, which is defined in terms of an auxiliary function, Eq':

$$\mathsf{Eq} :: \Omega \to \Omega$$

$$\mathsf{Eq}[t] \quad = \quad (\mathsf{Eq}'[t])[t]$$

$$\mathsf{Eq}' :: \Omega \to (\Omega \to \Omega)$$

$$\begin{array}{lcl}
\mathsf{Eq}'[\mathsf{Int}] & = & \lambda t{::}\Omega.t \\
\mathsf{Eq}'[\mathsf{Float}] & = & \lambda t{::}\Omega.t \\
\mathsf{Eq}'[\mathsf{Unit}] & = & \lambda t{::}\Omega.t \\
\mathsf{Eq}'[\mathsf{Prod}(t_1, t_2)] & = & \lambda t{::}\Omega.\mathsf{Eq}'[t_2](\mathsf{Eq}'[t_1]\ t) \\
\mathsf{Eq}'[\mathsf{Arrow}(t_1, \cdots, t_k, t')] & = & \lambda t{::}\Omega.\mathsf{Void} \\
\mathsf{Eq}'[\mathsf{Void}] & = & \lambda t{::}\Omega.\mathsf{Void}
\end{array}$$

The Eq' function returns the identity function on Ω if its argument is an equality type. Otherwise, Eq' returns the function that maps every monotype to Void. Therefore, $(\mathsf{Eq}[\mu])[\mu]$ returns μ whenever μ is an equality type, and Void otherwise. In essence, Eq' serves as an "if-then-else" construct that checks to see if μ is an equality type, and if so returns μ.

Now we can write the polymorphic equality function as follows:

$$\begin{array}{lcl}
\mathsf{peq}[\mathsf{Int}] & = & \lambda[x_1{:}\mathsf{int}, x_2{:}\mathsf{int}].\ \mathtt{eqint}(x_1, x_2) \\
\mathsf{peq}[\mathsf{Float}] & = & \lambda[x_1{:}\mathsf{float}, x_2{:}\mathsf{float}].\ \mathtt{eqfloat}(x_1, x_2) \\
\mathsf{peq}[\mathsf{Unit}] & = & \lambda[x_1{:}\mathsf{unit}, x_2{:}\mathsf{unit}].\ 1 \\
\mathsf{peq}[\mathsf{Prod}(t_a, t_b)] & = & \lambda\,[x_1{:}T(\mathsf{Prod}(t_a, t_b)), x_2{:}T(\mathsf{Prod}(t_a, t_b))]. \\
 & & \quad \mathtt{if0}\ \mathsf{peq}[t_a][\pi_1\,x_1, \pi_1\,x_2]\ \mathtt{then}\ 0\ \mathtt{else} \\
 & & \quad\quad \mathsf{peq}[t_b][\pi_2\,x_1, \pi_2\,x_2] \\
\mathsf{peq}[\mathsf{Arrow}(t_1, \cdots, t_k, t)] & = & \lambda\,[x_1{:}\mathsf{void}, x_2{:}\mathsf{void}].0
\end{array}$$

and the term can be assigned the following type:

$$\forall t{::}\Omega.[T(\mathsf{Eq}[t]), T(\mathsf{Eq}[t])] \to \mathsf{int}$$

Consequently, $\mathtt{peq}[\mu][e_1, e_2]$ is well-formed only if e_1 and e_2 have type $T(\mu)$ and μ is an equality type. In particular, the expression

$$\mathtt{peq}[\mathsf{Arrow}([\mathsf{Int}], \mathsf{Int})][\lambda x{:}\mathsf{int}.x, \lambda x{:}\mathsf{int}]$$

is ill-typed because `peq` applied to an Arrow constructor has type $[\mathsf{void}, \mathsf{void}] \to \mathsf{Int}$ and thus cannot be applied to the two functions of type $\mathsf{int} \to \mathsf{int}$. The encoding is not entirely satisfactory because `peq` can be applied to an Arrow constructor. However, the resulting expression can only be applied to arguments of type void. Since there are no closed values of type void, the resulting expression can never be invoked. Thus, an optimizer can safely replace the Arrow-clause of `peq` with some polymorphic constant (e.g., `error`).

This encoding suggests the following type translation for SML: Wrap each occurrence of an equality type variable with the Eq constructor function.

$$\begin{array}{rcl} |t| & = & t \\ |''t| & = & \mathsf{Eq}[''t] \\ \cdots & & \\ |\forall t_1, \cdots, t_n, ''t_1, \cdots, ''t_m.\tau| & = & \forall t_1, \cdots, t_n, ''t_1, \cdots, ''t_m.T(|\tau|) \end{array}$$

However, when instantiating a polymorphic function we must use an auxiliary translation that does not wrap the equality variables with Eq:

$$\begin{array}{rcl} ||t|| & = & t \\ ||''t|| & = & ''t \\ \cdots & & \end{array}$$

This auxiliary translation is needed because the translation above does not commute with substitution of equality types for equality type variables (i.e., an extra "Eq" gets wrapped around each equality type variable). Hence, the translation of polymorphic instantiation becomes:

$$(\mathbf{tapp})\ \frac{\begin{array}{c}\Delta \vdash \tau_1 \quad \cdots \quad \Delta \vdash \tau_n \quad \Delta \vdash ''\tau_1 \quad \cdots \quad \Delta \vdash ''\tau_m \\ \Delta; \Gamma \vdash v : \forall t_1, \cdots, t_n, ''t_1, \cdots, ''t_m.\tau\end{array}}{\Delta; \Gamma \vdash \begin{array}{c} v[\tau_1, \cdots, \tau_n, ''\tau_1, \cdots, ''\tau_m] : \{\tau_1/t_1, \cdots, \tau_n/t_n, ''\tau_1/''t_1, \cdots, ''\tau_m/''t_m\} \Rightarrow \\ v : [||\tau_1||, \cdots, ||\tau_n||, ||''\tau_1||, \cdots, ||''\tau_m||]\end{array}}$$

It is easy to verify that the type translation commutes with type substitution,

$$\{||\tau||/''t\}|\tau'| \equiv |\{\tau'/''t\}\tau'|,$$

and thus the resulting term has the appropriate type.

In this fashion, Typerec can be used to encode equality types and other type classes, whereas `typerec` can be used to implement the methods (i.e., `peq`) of the class. The information encoded in the type class can be used by a compiler to eliminate unneeded cases within methods.

5.4.3 Communication Primitives

Ohori and Kato give an extension of ML with primitives for communication in a distributed, heterogeneous environment [100]. Their extension has two essential features: one is a mechanism for generating globally unique names ("handles" or "capabilities") that are used as proxies for functions provided by servers. The other is a method for representing arbitrary values in a form suitable for transmission through a network. Integers are considered transmissible, as are pairs of transmissible values, but functions cannot be transmitted (due to the heterogeneous environment) and are thus represented by proxy. These proxies are associated with their functions by a name server that may be contacted through a primitive addressing scheme. In this section I sketch how a variant of Ohori and Kato's representation scheme can be implemented using dynamic type dispatch.

To accommodate Ohori and Kato's primitives, I extend λ_i^{ML}-Rep with a primitive constructor Proxy of kind $\Omega \to \Omega$ and a corresponding type constructor $\mathsf{proxy}(\sigma)$, linked by the equation $T(\mathsf{Proxy}(\mu)) \equiv \mathsf{proxy}(T(\mu))$. The $\mathsf{Typerec}$ and `typerec` primitives are extended in the obvious way to account for constructors of the form $\mathsf{Proxy}(\mu)$.

Next, I add primitives `proxy` and `rpc` with the following types:

$$\texttt{proxy} \;:\; \forall t_1, t_2{::}\Omega.(T(\mathsf{Tran}[t_1]) \to T(\mathsf{Tran}[t_2])) \to T(\mathsf{Tran}[\mathsf{Arrow}(t_1, t_2)])$$

$$\texttt{rpc} \;:\; \forall t_1, t_2{::}\Omega.(T(\mathsf{Tran}[\mathsf{Arrow}(t_1, t_2)])) \to T(\mathsf{Tran}[t_1]) \to T(\mathsf{Tran}[t_2])$$

where Tran is a constructor coded using $\mathsf{Typerec}$ as follows:

$$\mathsf{Tran} :: \Omega \to \Omega$$

$$\begin{array}{lcl}
\mathsf{Tran}[\mathsf{Int}] & = & \mathsf{Int} \\
\mathsf{Tran}[\mathsf{Float}] & = & \mathsf{Float} \\
\mathsf{Tran}[\mathsf{Unit}] & = & \mathsf{Unit} \\
\mathsf{Tran}[\mathsf{Prod}(t_1, t_2)] & = & \mathsf{Prod}(\mathsf{Tran}[t_1], \mathsf{Tran}[t_2]) \\
\mathsf{Tran}[\mathsf{Arrow}(t_1, t_2)] & = & \mathsf{Proxy}(\mathsf{Arrow}(t_1, t_2)) \\
\mathsf{Tran}[\mathsf{Proxy}(t)] & = & \mathsf{Proxy}(t)
\end{array}$$

The constructor $\mathsf{Tran}[\mu]$ maps μ to a constructor where each arrow is wrapped by a Proxy constructor. Thus, values of type $T(\mathsf{Tran}[\mu])$ do not contain functions and are therefore *transmissible.*

The `proxy` primitive takes a function between transmissible values, generates a new, globally unique proxy and tells the name server to associate that proxy with the function. For example, the proxy might consist of the machine's name paired with the address of the function. Conversely, the `rpc` operation takes a proxy of a function and a transmissible argument value. Then, the operation contacts the name sever to find the function corresponding to the proxy. When the function is found, the argument value is sent to the appropriate machine. Then, the function associated with the proxy is applied to the argument, and the result of the function is transmitted back as the result of the operation. Thus, `proxy` maps a function on transmissible representations to a transmissible representation of the function, whereas `rpc` maps a transmissible representation of a function to a function on transmissible representations.

The goal of Ohori and Kato's compilation was to provide transparent communication. That is, given any function `f` of type $T(\mu) \to T(\mu')$, their goal was to be able to transmit a representation of `f` to a remote site. We cannot obtain a proxy for `f` directly, because proxies require functions that take and return transmissible representations. Therefore, the key to transparent communication is a function `marshal` that coerces `f` to take and return transmissible values. In general, we want marshal to take any value and convert it to a transmissible representation.

I can write `marshal` using `typerec`. The definition requires a dual function, `unmarshal`, to accommodate function arguments[4].

$$\texttt{marshal} : \forall t{::}\Omega.T(t) \to T(\mathsf{Tran}[t])$$

$$\begin{array}{lcl}
\texttt{marshal}[\mathsf{Int}] & = & \lambda x{:}\mathsf{int}.x \\
\texttt{marshal}[\mathsf{Float}] & = & \lambda x{:}\mathsf{float}.x \\
\texttt{marshal}[\mathsf{Unit}] & = & \lambda x{:}\mathsf{unit}.x \\
\texttt{marshal}[\mathsf{Prod}(t_1,t_2)] & = & \lambda x{:}T(\mathsf{Prod}(t_1,t_2)). \\
 & & \quad \langle \texttt{marshal}[t_1](\pi_1\ x), \texttt{marshal}[t_2](\pi_2\ x)\rangle \\
\texttt{marshal}[\mathsf{Arrow}(t_1,t_2)] & = & \lambda f{:}T(\mathsf{Arrow}(t_1,t_2)). \\
 & & \quad \texttt{proxy}[t_1][t_2] \\
 & & \quad\ (\lambda x{:}T(\mathsf{Tran}[t_1]).\texttt{marshal}[t_2](f\ (\texttt{unmarshal}[t_1]\ x))) \\
\texttt{marshal}[\mathsf{Proxy}(t)] & = & \lambda x{:}T(\mathsf{Proxy}(t)).x
\end{array}$$

$$\texttt{unmarshal} : \forall t{::}\Omega.T(\mathsf{Tran}[t]) \to T(t)$$

$$\begin{array}{lcl}
\texttt{unmarshal}[\mathsf{Int}] & = & \lambda x{:}\mathsf{int}.x \\
\texttt{unmarshal}[\mathsf{Float}] & = & \lambda x{:}\mathsf{float}.x \\
\texttt{unmarshal}[\mathsf{Unit}] & = & \lambda x{:}\mathsf{unit}.x \\
\texttt{unmarshal}[\mathsf{Prod}(t_1,t_2)] & = & \lambda x{:}T(\mathsf{Tran}[\mathsf{Prod}(t_1,t_2)]). \\
 & & \quad \langle \texttt{unmarshal}[t_1](\pi_1\ x), \texttt{unmarshal}[t_2](\pi_2\ x)\rangle \\
\texttt{unmarshal}[\mathsf{Arrow}(t_1,t_2)] & = & \lambda f{:}T(\mathsf{Proxy}(\mathsf{Arrow}(\mathsf{Tran}[t_1],\mathsf{Tran}[t_2]))).\lambda x{:}T(t_1). \\
 & & \quad \texttt{unmarshal}[t_2](\texttt{rpc}[t_1][t_2]\ f\ (\texttt{marshal}[t_1]\ x)) \\
\texttt{unmarshal}[\mathsf{Proxy}(t)] & = & \lambda x{:}T(\mathsf{Proxy}(t)).x
\end{array}$$

[4] Technically, I must calculate `marshal` and `unmarshal` with one `typerec` and return a tuple containing the two functions.

At arrow types, `marshal` converts the given function to one that takes and returns transmissible types, and then allocates a new proxy for the resulting function. Conversely, `unmarshal` takes a proxy and a marshaled argument, performs an `rpc` on the proxy, and then unmarshals the result.

With `marshal` and `unmarshal`, I can dynamically convert a value to and from its transmissible representation. In effect, these terms *reify* the stub compilers of traditional RPC systems (e.g., the Mach Interface Generator for Mach RPC [70, 123]). Similarly, we can code general-purpose `print` and `read` routines within λ_i^{ML}, in order to achieve the easy input/output of languages like Lisp and Scheme.

5.5 Related Work

Peyton Jones and Launchbury suggested an approach to unboxed integers and reals in the context of a lazy language [75]. However, they restricted unboxed types from instantiating type variables. A similar idea was recently proposed by Ohori [101] to compile polymorphic languages such as SML.

Leroy suggested the coercion based approach to allow unrestricted instantiation of type variables [81], and later, Poulsen extended his work to accommodate unboxed datatypes that do not "escape" [102]. Henglein and Jørgensen examined techniques for eliminating coercions at compile-time. Shao and Appel [110, 108] took the ideas of Leroy and extended them to the full Standard ML language. Thiemann extended the work of Leroy to keep some values unboxed even within polymorphic functions [118]. None of these approaches supports unboxed mutable data, or generally unboxed datatypes. Furthermore, they do not address type classes, marshaling, or garbage collection.

Of a broadly similar nature is the work on "soft" type systems [64, 7, 29, 132]. Here, ML-style type inference or set constraints are used to eliminate type-tag checks in dynamically typed languages such as Scheme.

Morrison, et al. [97] described an implementation of Napier that passed types at run time to determine the behavior of polymorphic operations. However, the actual transformations performed were not described and there was little or no analysis of the typing properties or performance of the resulting code. The work of Ohori on compiling record operations [99] is similarly based on a type-passing interpretation and provided much of the inspiration of this work. Type passing was also used by Aditya and Caro in an implementation of Id, so that instantiations of polymorphic types could be reconstructed for debugging purposes [5].

Jones [72, 71] has proposed a general framework for passing data derived from types to "qualified" polymorphic operations, called *evidence passing*. He shows how evidence passing can be used to implement Haskell-style type classes, generalizing the earlier work of Wadler and Blott [122]. He also shows how Ohori-style record calculi can be implemented with evidence passing. Jones's approach subsumes type passing in that functions or types or any evidence derived from qualified types could, in principle, be passed to polymorphic operations. However, qualified types represent predicates on types, whereas the type system of λ_i^{ML} supports computations that transform types. For example, it is not possible to express the transmissible representation or a flattened representation of a type in Jones's framework.

Recently, Duggan and Ophel [38] and Thatte [116] have independently suggested semantics for type classes that are similar in spirit to my proposal. In one sense, these proposals do a better job of enforcing type classes, since they restrict the kinds of type variables. However, like Jones's qualified types, neither of these approaches can express transformations on types.

Dubois, Rouaix, and Weis formulated an approach to polymorphism dubbed "extensional polymorphism" [37][5]. The goal was to provide a framework to type check *ad hoc* operators like polymorphic equality. As with λ_i^{ML}, their formulation requires that some types be passed at runtime and be examined using what amounts to a structural induction elimination form. Their approach is fairly general since it is not restricted to primitive recursion over monotypes. However, type checking for the language is in general undecidable and type errors can occur at run time. Furthermore, like the approaches to type classes, there is no facility for transforming types.

Marshalling in languages with abstract or polymorphic types has been the subject of much research [85, 86, 84, 66, 27, 4, 100, 77]. The solution I propose does not easily extend to user-defined abstract types (as with Herlihy and Liskov [66]). However, none of these previous approaches are able to express the relationship between a value's type and its transmissible representation, whereas I am able to express this relationship as a constructor function (i.e., Tran).

[5] Originally, Harper and I termed the type analysis of λ_i^{ML} "intensional polymorphism".

Chapter 6

Typed Closure Conversion

In the previous chapters, I argued that types and dynamic type dispatch are important for *compiling* programming languages like Mini-ML. I showed that types can be used to direct compilation in choosing primitive operations, data structure layout, and calling conventions, and that types can direct a proof that a compiler is correct.

If we are to use types at run time for dynamic type dispatch, we must propagate type information all the way through the lowest levels of a compiler. This is one reason why *type-preserving* transformations, such as the translation from Mini-ML to λ_i^{ML}-Rep of Chapter 5, are so important.

In this chapter, I present a particularly important stage of compilation for functional programming languages known as *closure conversion*. To my knowledge, no one (besides Yasuhiko Minamide, Robert Harper and myself [92, 91]) has presented a type-preserving closure conversion phase, especially for type-passing polymorphic languages. Therefore, it is important to show that such a translation exists if I am to claim that my type-based implementation approach is viable. In this chapter, I show how to closure convert λ_i^{ML}-Rep using abstract closures. Minamide, Harper, and Morrisett provide further details on environment representations and how to represent closures [92, 91].

I begin by giving an overview of closure conversion and why it is an important part of functional language implementation. I then define a target language called λ_i^{ML}-Close, which is a variant of λ_i^{ML}-Rep that provides explicit facilities for constructing closures and their environments. Next, I give a type-directed and type-preserving closure transform from λ_i^{ML}-Rep to λ_i^{ML}-Close and prove that it is correct using the same methodology I used to compile Mini-ML to λ_i^{ML}-Rep.

6.1 An Overview of Closure Conversion

Standard operational models of programming languages based on the λ-calculus, such as the contextual semantics of Mini-ML and λ_i^{ML}-Rep, compute by substituting terms for variables in other terms. Substitution is expensive because it requires traversing and copying a term in order to find and replace all occurrences of the given variable. A well-known technique for mitigating these costs is to delay substitution until the binding of the variable is required during evaluation [80, 2, 1]. This is accomplished by pairing an open term with an environment that provides values for the free variables in the term. The open term may be thought of as immutable *code* that acts on the environment. Since the code is immutable, it can be generated once and shared among all instances of a function.

Closure conversion [105, 111, 33, 78, 76, 9, 124, 54] is a program transformation that achieves such a separation between code and data. Functions with free variables are replaced by code abstracting an extra environment parameter. Free variables in the body of the function are replaced by references to the environment. The abstracted code is "partially applied" to an explicitly constructed environment providing the bindings for these variables. This "partial application" of the code to its environment is, in fact, suspended until the function is actually applied to its argument; the suspended application is called a "closure", a data structure containing pure code and a representation of its environment.

The main ideas of closure conversion are illustrated by considering the following monomorphic ML program:

```
let val x = 1
    val y = 2
    val z = 3
    val f = λw. x + y + w
in
    f 100
end
```

The function `f` contains free variables `x` and `y`, but not `z`. We may eliminate the references to these variables from the body of `f` by abstracting an environment `env`, and by replacing `x` and `y` by references to the environment. In compensation, a suitable environment containing the bindings for `x` and `y` must be passed to `f` before it is applied. This leads to the following translation:

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv. λw. (π₁ env) + (π₂ env) + w) ⟨x,y⟩
in  f 100
end
```

References to x and y in the body of f are replaced by projections (field selections) π_1 and π_2 that access the corresponding component of the environment. Since the code for f is closed, it may be hoisted out of the enclosing definition and defined at the top-level. I ignore this "hoisting" phase and instead concentrate on the process of closure conversion.

In the preceding example, the environment contains bindings only for x and y, and is thus as small as possible. Since the body of f *could* contain an occurrence of z, it is also sensible to include z in the environment, resulting in the following code:

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv.  λw. (π₁ env) + (π₂ env) + w) ⟨x,y,z⟩
in
    f 100
end
```

In the above example I chose a "flat" (FAM-like [26]) representation of the environment as a record with one field for each variable. Alternatively, I could choose a "linked" (CAM-like [33]) representation where, for example, each binding is a separate "frame" attached to the front of the remaining bindings. This idea leads to the following translation:

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv. λw. (π₁(π₂(π₂ env))) + (π₁(π₂ env)) + w)
              ⟨z,⟨y,⟨x,⟨⟩⟩⟩⟩
in
    f 100
end
```

The linked representation facilitates sharing of environments, but at the expense of introducing link traversals proportional to the nesting depth of the variable in the environment. The linked representation can also support constant-time closure creation, but this requires reusing the current environment and can result in bindings in the environment for variables that do not occur free in the function (such as z above), leading to space leaks [109].

Closure conversion for a language like λ_i^{ML} where constructors are passed at run time is complicated by the fact that we must account for free *type* variables as well as free value variables within code. Furthermore, both value abstractions (λ-terms) and constructor abstractions (Λ-terms) induce the creation of closures.

As an example, consider the expression:

$$\lambda\texttt{x}{:}t_1\texttt{.}\ \texttt{(x}{:}t_1\texttt{,}\ \texttt{y}{:}t_2\texttt{,}\ \texttt{z:int)}$$

of type $t_1 \to (t_1 \times t_2 \times \texttt{int})$ where t_1 and t_2 are free type variables and y and z are free value variables of type t_2 and int respectively. After closure conversion, this expression is translated to the partial application

```
let val code =
      Λtenv :: Ω × Ω.
        λvenv : T(π₁ tenv)×int.
          λx : T(π₁ tenv).(x, π₁ venv, π₂ venv)
in
    code (t₁,t₂) ⟨y,z⟩
end
```

The code abstracts type environment (tenv) and value environment (venv) arguments. The actual type environment, (t_1,t_2), is a constructor tuple with kind $\Omega \times \Omega$. The actual value environment, $\langle \texttt{y,z} \rangle$, is a tuple with type $T(t_2)\times\texttt{int}$. However, to keep the code closed so that it may be hoisted and shared, all references to free type variables in the type of venv must come from tenv. Thus, we give venv the type $T(\pi_1\ \texttt{tenv})\times\texttt{int}$. Similarly, the code's argument x is given the type $T(\pi_1\ \texttt{tenv})$. Consequently, the code part of the closure is a closed expression of closed type σ, where

$$\sigma = \forall\texttt{tenv}{::}\Omega \times \Omega.\ T(\pi_1\ \texttt{tenv})\times\texttt{int}\to T(\pi_1\ \texttt{tenv})\to(T(\pi_1\ \texttt{tenv})\times T(\pi_2\ \texttt{tenv})\times\texttt{int})$$

It is easy to check that the entire expression has type $t_1 \to (t_1 \times t_2 \times \texttt{int})$, and thus the type of the original function is preserved.

6.2 The Target Language: λ_i^{ML}-Close

The target language of the closure conversion translation is called λ_i^{ML}-Close. The syntax of this language is given in Figure 6.1. The constructors of the language are similar to those of λ_i^{ML}-Rep, except that I have added unit, products, and projections for building constructor environments. To the types, I have added a code type, $\texttt{code}(t{::}\kappa, \sigma_1, \sigma_2)$, corresponding to both value-abstraction code ($\texttt{vcode}$) and type-abstraction code ($\texttt{tcode}$), where t is the abstracted type environment and σ_1 is the abstracted value environment. If σ_2 is an arrow-type, then the code type describes code for a value abstraction, and if σ_2 is a $\forall$-type, then the code type describes code for a type abstraction.

The best way to understand these new constructs is to relate them informally to standard λ_i^{ML}-Rep constructs as follows:

$$\texttt{code}(t{::}\kappa, \sigma, \sigma') \approx \forall t{::}\kappa.\sigma \to \sigma'$$

$$\begin{array}{rcl}
\texttt{vcode}[t{::}\kappa, x{:}\sigma, x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].e & \approx & \Lambda t{::}\kappa.\lambda x{:}\sigma.\lambda[x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].e \\
\texttt{tcode}[t{::}\kappa, x{:}\sigma, t'{::}\kappa'].e & \approx & \Lambda t{::}\kappa.\lambda x{:}\sigma.\Lambda t'{::}\kappa'.e \\
\langle\langle e_1, \mu, e_2\rangle\rangle & \approx & (e_1\ [\mu])\ e_2
\end{array}$$

Code terms abstract a type environment and a value environment. In the case of value code, I also abstract a set of k value arguments; for type code, I abstract an additional type argument. I have added a special closure form to terms, $\langle\langle e_1, \mu, e_2\rangle\rangle$, where e_1 is the code of the closure, μ is the type environment, and e_2 is the value environment. Closure terms represent the delayed partial application of code to its environments.

Technically, I need to provide code and closure forms at the constructor level as well as the term level. However, doing so requires redefining an appropriate notion of constructor equivalence and reduction in the presence of constructor code and closures. This in turn requires reproving properties, such as strong-normalization and confluence for reduction of constructors. To avoid this complexity and to simplify the presentation, I will use standard λ-abstractions and partial applications to represent code and closures at the constructor level.

The constructor formation and equivalence rules are essentially the same as in λ^{ML}-REP (See Figure 5.2) except for the addition of rules pertaining to constructor-level products. I add formation rules for products as follows:

$$\Delta \vdash () :: 1 \qquad \frac{\Delta \vdash \mu_1 :: \kappa_1 \qquad \Delta \vdash \mu_2 :: \kappa_2}{\Delta \vdash \langle \mu_1, \mu_2 \rangle :: \kappa_1 \times \kappa_2} \qquad \frac{\Delta \vdash \mu :: \kappa_1 \times \kappa_2}{\Delta \vdash \pi_i\, \mu :: \kappa_i}$$

I add both β and η-like rules governing equivalences of products as follows:

$$\frac{\Delta \vdash \mu :: 1}{\Delta \vdash \mu \equiv ()}$$

$$\frac{\Delta \vdash \mu_1 :: \kappa_1 \qquad \Delta \vdash \mu_2 :: \kappa_2}{\Delta \vdash \pi_i\, \langle \mu_1, \mu_2 \rangle \equiv \mu_i :: \kappa_i} \qquad \frac{\Delta \vdash \mu :: \kappa_1 \times \kappa_2}{\Delta \vdash \langle \pi_1\, \mu, \pi_2\, \mu \rangle \equiv \mu :: \kappa_1 \times \kappa_2}$$

(kinds)	κ	$::=$	$\Omega \mid 1 \mid \kappa_1 \times \kappa_2 \mid \kappa_1 \to \kappa_2$
(constructors)	μ	$::=$	$t \mid \mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Unit} \mid \mathsf{Prod}(\mu_1, \mu_2) \mid \mathsf{Arrow}([\mu_1, \cdots, \mu_k], \mu) \mid$ $() \mid (\mu_1, \mu_n) \mid \pi_1\ \mu \mid \pi_2\ \mu \mid \lambda t{::}\kappa.\ \mu \mid \mu_1\, \mu_2 \mid$ $\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)$
(types)	σ	$::=$	$T(\mu) \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{unit} \mid \langle \sigma_1 \times \sigma_2 \rangle \mid [\sigma_1, \cdots, \sigma_k] \to \sigma \mid$ $\forall t{::}\kappa.\sigma \mid \mathsf{code}(t{::}\kappa, \sigma_1, \sigma_2)$
(expressions)	e	$::=$	$x \mid i \mid f \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid$ $\mathtt{vcode}[t{::}\kappa, x{:}\sigma, x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].e \mid \mathtt{tcode}[t_1{::}\kappa, x{:}\sigma, t_2{::}\kappa_2].e \mid$ $\langle\langle e_1, \mu, e_2 \rangle\rangle \mid e\,[e_1, \cdots, e_n] \mid e[\mu] \mid$ $\mathtt{eqint}(e_1, e_2) \mid \mathtt{eqfloat}(e_1, e_2) \mid \mathtt{if0}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \mid$ $\mathtt{typerec}\ \mu\ \mathtt{of}\ [t.\sigma](e_i; e_f; e_u; e_p; e_a)$

Figure 6.1: Syntax of λ_i^{ML}-Close

I also add appropriate congruences for both product and projection formation (not shown here).

The type formation and equivalence rules are the same as in λ_i^{ML}-Rep, with the addition of rules governing code types. The code type formation rule is similar to the one governing $\forall$:

$$\frac{\Delta \uplus \{t{::}\kappa\} \vdash \sigma_1 \qquad \Delta \uplus \{t{::}\kappa\} \vdash \sigma_2}{\Delta \vdash \mathsf{code}(t{::}\kappa, \sigma_1, \sigma_2)}$$

The interesting term formation rules pertain to code and closures and are as follows:

$$\frac{\begin{array}{c}\Delta \uplus \{t{::}\kappa\} \vdash \sigma \\ \Delta \uplus \{t{::}\kappa\} \vdash \sigma_1 \quad \cdots \quad \Delta \uplus \{t{::}\kappa\} \vdash \sigma_k \\ \Delta \uplus \{t{::}\kappa\}; \Gamma \uplus \{x{:}\sigma, x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k\} \vdash e : \sigma'\end{array}}{\Delta; \Gamma \vdash \mathtt{vcode}[t{::}\kappa, x{:}\sigma, x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].e : \mathsf{code}(t{::}\kappa, \sigma, [\sigma_1, \cdots, \sigma_k] \to \sigma')}$$

$$\frac{\begin{array}{c}\Delta \uplus \{t{::}\kappa\} \vdash \sigma \qquad \Delta \uplus \{t{::}\kappa, t'{::}\kappa'\} \vdash \sigma' \\ \Delta \uplus \{t{::}\kappa, t'{::}\kappa'\}; \Gamma \uplus \{x{:}\sigma\} \vdash e : \sigma'\end{array}}{\Delta; \Gamma \vdash \mathtt{tcode}[t{::}\kappa, x{:}\sigma, t'{::}\kappa'].e : \mathsf{code}(t{::}\kappa, \sigma, \forall t'{::}\kappa'.\sigma')}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{code}(t{::}\kappa, \sigma, \sigma') \qquad \Delta \vdash \mu :: \kappa \qquad \Delta; \Gamma \vdash e_2 :: \{\mu/t\}\sigma}{\Delta; \Gamma \vdash \langle\langle e_1, \mu, e_2 \rangle\rangle :: \{\mu/t\}\sigma'}$$

The values, contexts, instructions, and rewriting rules for constructors are standard, except for the addition of products (which is straightforward). If, at the constructor level, I introduced code and closures instead of λ-abstractions, then we would consider these constructs to be values (assuming the components of the closures are values). Application of a closure to a constructor value would proceed by substituting both the environment of the closure and the argument for the abstracted constructor variables in the code. These issues are demonstrated at the term level.

The values, contexts, and instructions for terms are standard except for the following changes: first, I consider both `vcode` and `tcode` terms to be values, as well as closures containing value components:

$$\begin{array}{llll}(\text{values}) & v & ::= & \cdots \mid \texttt{vcode}[t{::}\kappa, x{:}\sigma, x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].e \mid \\ & & & \texttt{tcode}[t{::}\kappa, x{:}\sigma, t'{::}\kappa'].\ \mid \langle\langle v, u, v' \rangle\rangle\end{array}$$

I extend evaluation contexts so that closure components are evaluated in a left-to-right fashion as follows:

$$(\text{contexts}) \quad E \quad ::= \quad \cdots \mid \langle\langle E, \mu, e \rangle\rangle \mid \langle\langle v, u, E \rangle\rangle$$

In the instructions, I replace application of abstractions to values with applications of closures to values. I also add an instruction to evaluate the constructor component of a closure. This yields instructions of the form

$$\begin{array}{llll}(\text{instructions}) & I & ::= & \cdots \mid \langle\langle v, U[J], e \rangle\rangle \mid \langle\langle v', u, v \rangle\rangle\,[v_1, \cdots, v_k] \mid \\ & & & \langle\langle v', u, v \rangle\rangle[u']\end{array}$$

with the restriction that the first component of a closure must be an appropriate code term, according to the application (see below).

Finally, the rewriting rules for both value and constructor application are as follows:

$$\begin{array}{l}E[\langle\langle \texttt{vcode}[t{::}\kappa, x{:}\sigma, x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].e, u, v \rangle\rangle\,[v_1, \cdots, v_k]] \longmapsto \\ \qquad\qquad E[\{u/t, v/x, v_1/x_1, \cdots, v_k/x_k\}e]\end{array}$$

$$E[\langle\langle \texttt{tcode}[t{::}\kappa, x{:}\sigma, t'{::}\kappa'].e, u, v \rangle\rangle\,[u']] \longmapsto E[\{u/t, v/x, u'/t'\}e]$$

In each case, we open the closure and extract the code, type environment, and value environment. We then substitute the environments and the argument(s) for the appropriate variables in the code.

It is straightforward to show that the static semantics of λ_i^{ML}-Close, as with λ_i^{ML} and λ_i^{ML}-Rep, is sound with respect to the operational semantics and that type-checking λ_i^{ML}-Close terms is decidable.

6.3 The Closure Conversion Translation

The closure conversion translation is broken into a constructor translation, a type translation, and a term translation. I use the source kind and type judgments to define these translations, but I augment the judgments with additional structure to determine certain details in the translation.

Throughout the translation, I use n-tuples at both the constructor and term levels as abbreviations for right-associated binary products, terminated with a unit. For instance, I use $(\kappa_1 \times \kappa_2 \times \cdots \times \kappa_n)$ to abbreviate the kind $\kappa_1 \times (\kappa_2 \times (\cdots \times (\kappa_n \times 1) \cdots))$ and $(\mu_1, \mu_2, \cdots, \mu_n)$ to abbreviate the constructor $(\mu_1, (\mu_2, (\cdots (\mu_n, ()) \cdots)))$. Correspondingly, I use $\#1(\mu)$ as an abbreviation for $\pi_1\, \mu$ and $\#i(\mu)$ as an abbreviation for $\#(i-1)(\pi_2\, \mu)$ when $i > 1$.

6.3.1 The Constructor and Type Translations

I begin the translation by considering closure conversion of λ_i^{ML}-Rep constructors. Constructor translation judgments are of the form

$$\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \kappa \Rightarrow \mu',$$

where $\Delta_{\text{env}} \uplus \Delta_{\text{arg}} \vdash \mu :: \kappa$ is derivable from the constructor formation rules of λ_i^{ML}-Rep, and μ' is a λ_i^{ML}-Close constructor. The axioms and inference rules that allow us to derive this judgment are given in Figure 6.2.

In the constructor translation judgment, I split the kind assignment into two pieces: Δ_{env} and Δ_{arg}. The Δ_{arg} component contains a kind assignment for the type variable bound by the nearest enclosing λ-abstraction (if any). The Δ_{env} component contains a kind assignment for the other type variables in scope. The translation maps a type variable found in Δ_{arg} to itself, but maps type variables in Δ_{env} to a projection from a type environment data structure. This data structure is assumed to be bound to the distinguished target variable t_{env}. Hence, I assume that t_{env} does not occur in the domain of Δ_{arg}. The projections assume that the order of bindings in the kind assignment does not change, so I consider Δ_{env} to be an ordered sequence, binding variables to kinds.

The rest of the translation is straightforward with the exception of λ-abstractions, which we must closure-convert. In this case, I generate a piece of code of the form $\lambda t_{\text{env}}::\kappa_{\text{env}}.\, \lambda t::\kappa_1.\, \mu'$, which abstracts both a type environment (t_{env}) and an argument (t). I also generate an environment, μ_{env} (discussed below). The code is obtained by choosing a new kind assignment, Δ'_{env}, to replace Δ_{env}, and by replacing Δ_{arg} with $\{t::\kappa_1\}$ in the translation of the body of the abstraction. Choosing the new kind assignment Δ'_{env} corresponds to deciding which variables will be preserved in the environment of the closure. Therefore, Δ'_{env} must be a subset of the bindings contained in $\Delta_{\text{env}} \uplus \Delta_{\text{arg}}$, and Δ'_{env} must contain bindings for all of the free type variables in the abstraction.

$$(\textbf{var-arg})\ \ \Delta_{\text{env}}; \{t{::}\kappa\} \vdash t :: \kappa \Rightarrow t \qquad\qquad (\textbf{unit})\ \ \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Unit} :: \Omega \Rightarrow \mathsf{Unit}$$

$$(\textbf{var-env})\ \ \{t_1{::}\kappa_1, \cdots, t_n{::}\kappa_n\}; \Delta_{\text{arg}} \vdash t_i :: \kappa_i \Rightarrow \#i(t_{\text{env}})$$

$$(\textbf{int})\ \ \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Int} :: \Omega \Rightarrow \mathsf{Int} \qquad\qquad (\textbf{float})\ \ \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Float} :: \Omega \Rightarrow \mathsf{Float}$$

$$(\textbf{prod})\ \ \frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_1 :: \Omega \Rightarrow \mu_1' \quad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_2 :: \Omega \Rightarrow \mu_2'}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Prod}(\mu_1, \mu_2) :: \Omega \Rightarrow \mathsf{Prod}(\mu_1', \mu_2')}$$

$$(\textbf{arrow})\ \ \frac{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_1 :: \Omega \Rightarrow \mu_1' \quad \cdots \quad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_k :: \Omega \Rightarrow \mu_k' \\ \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \Omega \Rightarrow \mu'\end{array}}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Arrow}([\mu_1, \cdots, \mu_k], \mu) :: \Omega \Rightarrow \mathsf{Arrow}([\mu_1', \cdots, \mu_k'], \mu')}$$

$$(\textbf{fn})\ \ \frac{\Delta_{\text{env}}'; \{t{::}\kappa_1\} \vdash \mu :: \kappa_2 \Rightarrow \mu' \qquad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash_{\text{env}} \Delta_{\text{env}}' \Rightarrow \mu_{\text{env}}}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \lambda t{::}\kappa_1.\, \mu :: \kappa_1 \to \kappa_2 \Rightarrow (\lambda t_{\text{env}}{::}|\Delta_{\text{env}}'|.\, \lambda t{::}\kappa_1.\, \mu')\ \mu_{\text{env}}}$$

$$(\textbf{app})\ \ \frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_1 :: \kappa_1 \to \kappa_2 \Rightarrow \mu_1' \qquad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_2 :: \kappa_1 \Rightarrow \mu_2'}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_1\ \mu_2 :: \kappa_2 \Rightarrow \mu_1\ \mu_2'}$$

$$(\textbf{trec})\ \ \frac{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \Omega \Rightarrow \mu' \quad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_i, \mu_f, \mu_u :: \kappa \Rightarrow \mu_i', \mu_f', \mu_u' \\ \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_p :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \Rightarrow \mu_p' \\ \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu_a :: \Omega_1 \to \cdots \to \Omega_k \to \Omega \to \kappa_1 \to \cdots \to \kappa_k \to \kappa \to \kappa \Rightarrow \mu_a'\end{array}}{\begin{array}{l}\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a) :: \kappa \Rightarrow \\ \qquad \mathsf{Typerec}\ \mu'\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a')\end{array}}$$

$$(\textbf{env})\ \ \frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash t_1 :: \kappa_1 \Rightarrow \mu_1 \quad \cdots \quad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash t_n :: \kappa_n \Rightarrow \mu_n}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash_{\text{env}} \{t_1{::}\kappa_1, \cdots, t_n{::}\kappa_n\} \Rightarrow (\mu_1, \cdots, \mu_n)}$$

Figure 6.2: Closure Conversion of Constructors

I construct the environment of the closure using the auxiliary judgment

$$\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash_{\text{env}} \Delta'_{\text{env}} \Rightarrow \mu.$$

With the **env** rule, I create an environment corresponding to Δ'_{env} by extracting the value corresponding to each variable in the domain of Δ'_{env}, and then packing these values in order into a tuple. If $\Delta'_{\text{env}} = \{t_1{::}\kappa_1, \cdots, t_n{::}\kappa_n\}$, then the kind of the resulting environment is $(\kappa_1 \times \cdots \times \kappa_n)$, which I abbreviate as $|\Delta'_{\text{env}}|$.

To translate types, I use judgments of the form $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma \Rightarrow \sigma'$. The translation maps λ_i^{ML}-Rep types to the same λ_i^{ML}-Close types, except that the injected constructors are converted via the constructor translation:

$$\frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \Omega \Rightarrow \mu'}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash T(\mu) \Rightarrow T(\mu')}$$

The type translation of a polytype extends the current argument assignment, Δ_{arg}, with the bound type variable during the translation of the body of the polytype:

$$\frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \uplus \{t{::}\kappa\} \vdash \sigma \Rightarrow \sigma'}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \forall t{::}\kappa.\sigma \Rightarrow \forall t{::}\kappa.\sigma'}$$

6.3.2 The Term Translation

The term translation for closure conversion mirrors the constructor translation, except that I must account for both free type variables and free value variables. Judgments in the translation are of the form

$$\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash e : \sigma \Rightarrow e'$$

where $\Delta_{\text{env}} \uplus \Delta_{\text{arg}}; \Gamma_{\text{env}} \uplus \Gamma_{\text{arg}} \vdash e : \sigma$ is a λ_i^{ML}-Rep term formation judgment and e' is a λ_i^{ML}-Close expression. The important axioms and inference rules that let us derive this judgment are given in Figure 6.3. The rest of the rules simply map λ_i^{ML}-Rep terms to their corresponding λ_i^{ML}-Close terms.

Like the constructor translation, the kind assignment and the type assignment are split into environment and argument components. The argument component assigns kinds/types to the variables of the nearest enclosing Λ or λ-expression (if any); the environment component assigns kinds/types to the other free variables in scope. As in the constructor case, I translate a variable occurring in Γ_{arg} to itself, whereas I translate a variable occurring in Γ_{env} to a projection from a distinguished variable, x_{env}. Again, the order of bindings in Γ_{env} is relevant to the translation.

The **abs** and **tabs** rules translate abstractions to closures consisting of code, a constructor environment (μ_{env}), and a value environment (e_{env}). The code components are

$$(\textbf{var-arg})\ \ \Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \{x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k\} \vdash x_i : \sigma_i \Rightarrow x_i$$

$$(\textbf{var-env})\ \ \Delta_{\text{env}}; \Delta_{\text{arg}}; \{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\}; \Gamma_{\text{arg}} \vdash x_i : \sigma_i \Rightarrow \#i(x_{\text{env}})$$

$$(\textbf{tapp})\ \ \frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \kappa \Rightarrow \mu' \qquad \Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash e : \forall t{::}\kappa.\sigma \Rightarrow e'}{\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash e\ [\mu] : \{\mu/t\}\sigma \Rightarrow e'\ [\mu']}$$

$$(\textbf{abs})\ \ \frac{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash_{\text{env}} \Delta'_{\text{env}} \Rightarrow \mu_{\text{env}} \qquad \Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash_{\text{env}} \Gamma'_{\text{env}} \Rightarrow e_{\text{env}} \\ \Delta'_{\text{env}}; \emptyset \vdash_{\text{env-type}} \Gamma'_{\text{env}} \Rightarrow \Gamma''_{\text{env}} \\ \Delta'_{\text{env}}; \emptyset \vdash \sigma_1 \Rightarrow \sigma'_1 \quad \cdots \quad \Delta'_{\text{env}}; \emptyset \vdash \sigma_k \Rightarrow \sigma'_k \\ \Delta'_{\text{env}}; \emptyset; \Gamma'_{\text{env}}; \{x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k\} \vdash e : \sigma \Rightarrow e'\end{array}}{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash \lambda[x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].\, e : [\sigma_1, \cdots, \sigma_k] \to \sigma \Rightarrow \\ \langle\langle \texttt{vcode}[t_{\text{env}} :: |\Delta'_{\text{env}}|, x_{\text{env}}{:}|\Gamma''_{\text{env}}|, x_1{:}\sigma'_1, \cdots, x_k{:}\sigma'_k].e', \mu_{\text{env}}, e_{\text{env}}\rangle\rangle\end{array}}$$

$$(\textbf{tabs})\ \ \frac{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash_{\text{env}} \Delta'_{\text{env}} \Rightarrow \mu_{\text{env}} \qquad \Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash_{\text{env}} \Gamma'_{\text{env}} \Rightarrow e_{\text{env}} \\ \Delta'_{\text{env}}; \emptyset \vdash_{\text{env-type}} \Gamma'_{\text{env}} \Rightarrow \Gamma''_{\text{env}} \\ \Delta'_{\text{env}}; \{t{::}\kappa\}; \Gamma'_{\text{env}}; \emptyset \vdash e : \sigma \Rightarrow e'\end{array}}{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash \Lambda t{::}\kappa.e : \forall t{::}\kappa.\sigma \Rightarrow \\ \langle\langle \texttt{tcode}[t_{\text{env}} :: |\Delta'_{\text{env}}|, x_{\text{env}}{:}|\Gamma''_{\text{env}}|, t{::}\kappa].e', \mu_{\text{env}}, e_{\text{env}}\rangle\rangle\end{array}}$$

Figure 6.3: Closure Conversion of Terms

constructed by choosing new kind and type assignments to cover the free variables of the abstraction.

The environment components are constructed using the auxiliary judgments

$$\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \Delta'_{\text{env}} \Rightarrow \mu_{\text{env}}$$

and

$$\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash_{\text{env}} \Gamma'_{\text{env}} \Rightarrow e_{\text{env}}.$$

The former judgment is obtained via the **env** constructor translation rule, whereas the latter judgment is obtained via the following term translation rule:

$$(\mathbf{env})\ \frac{\begin{array}{c}\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash x_1{:}\sigma_1 \Rightarrow e_1 \\ \cdots \\ \Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash x_n{:}\sigma_n \Rightarrow e_n\end{array}}{\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash \{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\} \Rightarrow \langle e_1, \cdots, e_n\rangle}$$

This rule translates a new type assignment, Γ'_{env}, by extracting the values corresponding to each variable in the domain of the assignment and placing the resulting values in a tuple. To obtain the type of the resulting environment data structure, I first translate all of the types in the range of Γ'_{env}:

$$(\textbf{env-type})\ \frac{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma_1 \Rightarrow \sigma'_1 \quad \cdots \quad \Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma_n \Rightarrow \sigma'_n}{\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash_{\text{env-type}} \{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\} \Rightarrow \{x_1{:}\sigma'_1, \cdots, x_n{:}\sigma'_n\}}$$

This process results in a λ_i^{ML}-Close type assignment $\Gamma''_{\text{env}} = \{x_1{:}\sigma'_1, \cdots, x_n{:}\sigma'_n\}$. The tuple environment data structure has the type $\langle \sigma'_1 \times \cdots \times \sigma'_n \rangle$, which I abbreviate as $|\Gamma''_{\text{env}}|$.

6.4 Correctness of the Translation

To prove the correctness of the closure conversion translation, I will establish suitable relations between source and target constructors and terms, and then show that a source construct is always related to its translation. I first examine the correctness of constructor translation and, then consider term translation.

6.4.1 Correctness of the Constructor Translation

It is clear that if $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \kappa \Rightarrow \mu'$, then $\Delta_{\text{env}} \uplus \Delta_{\text{arg}} \vdash \mu :: \kappa$. It is also fairly easy to show that we can always construct some translation of a well-formed constructor. I only need to show that we can always delay any reordering or strengthening of the kind assignment until we reach a use of an **abs** rule. Finally, it is also easy to show via induction on the translation that constructor closure conversion preserves kinds directly.

Lemma 6.4.1 (Kind Correctness) *If* $\Delta_{\rm env}; \Delta_{\rm arg} \vdash \mu :: \kappa \Rightarrow \mu'$, *then* $\{t_{\rm env}{::}|\Delta_{\rm env}|\} \uplus \Delta_{\rm arg} \vdash \mu' :: \kappa$.

I want to show that a constructor and its translation are suitably equivalent. I begin by establishing a set of kind-indexed simulation relations relating closed source and target constructors. At base-kind, the relation is given as follows:

$$\frac{\emptyset \vdash \mu \equiv u :: \Omega \qquad \emptyset \vdash \mu' \equiv u :: \Omega}{\mu \approx_\Omega \mu'}$$

Two closed constructors of kind Ω are related if they are definitionally equivalent to the same constructor value. That is, to determine whether a source and target constructor are related, we simply normalize the constructors and then syntactically compare them. (Recall that constructors of the source language are a subset of the constructors in the target language and the two languages coincide at the base kind Ω.) I logically extend the base relation to arrow kinds:

$$\frac{\mu_1 \approx_{\kappa_1} \mu_1' \text{ implies } \mu\,\mu_1 \approx_{\kappa_2} \mu'\,\mu_1'}{\mu \approx_{\kappa_1 \to \kappa_2} \mu'}$$

Let δ and δ' range over substitutions of type variables for closed, source and target constructors respectively. I extend the relation to substitutions indexed by kind assignments:

$$\frac{\mathit{Dom}(\delta) = \{t_1, \cdots, t_n\} \qquad \forall 1 \leq i \leq n. \delta(t_i) \approx_{\kappa_i} \#i(\mu)}{\delta \approx_{\{t_1::\kappa_1, \cdots, t_n::\kappa_n\}} \{t_{\rm env}{=}\mu\}}$$

Note that the distinguished type variable $t_{\rm env}$ is used in the target substitution. Finally, I relate pairs of substitutions $\delta_{\rm env}; \delta_{\rm arg}$ and $\delta'_{\rm env}; \delta'_{\rm arg}$ as follows:

$$\frac{\begin{array}{c}\delta_{\rm env} \approx_{\Delta_{\rm env}} \delta'_{\rm env} \\ \mathit{Dom}(\delta_{\rm arg}) = \mathit{Dom}(\Delta_{\rm arg}) = \mathit{Dom}(\delta'_{\rm arg}) \\ \forall t \in \mathit{Dom}(\Delta_{\rm arg}).\delta_{\rm arg}(t) \approx_{\Delta_{\rm arg}(t)} \delta'_{\rm arg}(t)\end{array}}{\delta_{\rm env}; \delta_{\rm arg} \approx_{\Delta_{\rm env};\Delta_{\rm arg}} \delta'_{\rm env}; \delta'_{\rm arg}}$$

The pairs of substitutions are related iff $\delta_{\rm env}$ and $\delta'_{\rm env}$ are related under $\Delta_{\rm env}$, and for any argument variable t, $\delta_{\rm arg}(t)$ and $\delta'_{\rm arg}(t)$ are related at $\Delta_{\rm arg}(t)$.

With these definitions in place, I can state and prove the correctness of the constructor translation. The first step is to show that constructing new environments from related constructors yields related environments.

Lemma 6.4.2 *If* $\delta_{\rm env}; \delta_{\rm arg} \approx_{\Delta_{\rm env};\Delta_{\rm arg}} \delta'_{\rm env}; \delta'_{\rm arg}$, $\Delta_{\rm env}; \Delta_{\rm arg} \vdash_{\rm env} \Delta'_{\rm env} \Rightarrow (\mu_1, \cdots, \mu_n)$, *where* $\Delta'_{\rm env} = \{t_1{::}\kappa_1, \cdots, t_n{::}\kappa_n\}$, *then* $\{t_1{=}\delta_{\rm env} \uplus \delta_{\rm arg}(t_1), \cdots, t_n{=}\delta_{\rm env} \uplus \delta_{\rm arg}(t_n)\} \approx_{\Delta'_{\rm env}} \{t_{\rm env}{=}\delta'_{\rm env} \uplus \delta'_{\rm arg}(\mu_1, \cdots, \mu_n)\}$.

Proof: I must show that $\delta_{\text{env}} \uplus \delta_{\text{arg}}(t_i) \approx_{\kappa_i} \#i(\delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_1, \cdots, \mu_n))$ for $1 \leq i \leq n$. Hence, it suffices to show $\delta_{\text{env}} \uplus \delta_{\text{arg}}(t_i) \approx_{\kappa_i} \delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_i)$. By an examination of the **env** rule, $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash t_i{::}\kappa_i \mu_i$. There are two cases to consider, depending on the rule used to produce μ_i: either t_i is translated under the **var-arg** rule or else t_i is translated under the **var-env** rule. In the former case, $\mu_i = t_i$. By assumption, $\delta_{\text{env}}; \delta_{\text{arg}} \approx_{\Delta_{\text{env}};\Delta_{\text{arg}}} \delta'_{\text{env}}; \delta'_{\text{arg}}$, thus $\delta_{\text{arg}}(t_i) \approx_{\kappa_i} \delta'_{\text{arg}}(t_i)$. In the latter case, we have $\mu_i = \#j(t_{\text{env}})$, $t_i = t'_j$, and Δ_{env} is of the form $\{t'_1{::}\kappa'_1, \cdots, t'_j{::}\kappa_i, \cdots, t'_m{::}\kappa'_m\}$. By assumption, $\delta_{\text{env}} \approx_{\Delta_{\text{env}}} \delta_{\text{arg}}$. Therefore, by the definition of the relation, for all $1 \leq k \leq m$, $\delta_{\text{env}} t'_k \approx_{\kappa_k} \delta'_{\text{env}} \#k(t_{\text{env}})$. In particular, since $t'_j = t_i$, $\delta_{\text{env}}(t_i) \approx_{\kappa_i} \#j(\delta'_{\text{env}}(t_{\text{env}}))$. Consequently, in either case we know that $\delta_{\text{env}} \uplus \delta_{\text{arg}}(t_i) \approx_{\kappa_i} \delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_i)$. □

Next, by induction on the derivation of a translation, I show that a constructor and its translation are related when we apply related substitutions. In particular, the translation of a constructor of base kind yields a constructor that is definitionally equivalent to that constructor.

Theorem 6.4.3 (Constructor Correctness) *If* $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \kappa \Rightarrow \mu'$ *and* $\delta_{\text{env}}; \delta_{\text{arg}} \approx_{\Delta_{\text{env}};\Delta_{\text{arg}}} \delta'_{\text{env}}; \delta'_{\text{arg}}$, *then* $\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu) \approx_{\kappa} \delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu')$.

Proof: By induction on the derivation of $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mu :: \kappa \Rightarrow \mu'$. The interesting cases, **var-arg**, **var-env**, **fn**, and **trec** are given below.

var-arg: $\Delta_{\text{env}}; \{t{::}\kappa\} \vdash t :: \kappa \Rightarrow t$. By assumption, $\delta_{\text{arg}}(t) \approx_{\kappa} \delta'_{\text{arg}}(t)$.

var-env: $\{t_1{::}\kappa_1, \cdots, t_n{::}\kappa_n\} \vdash t_i :: \kappa_i \Rightarrow \#i(t_{\text{env}})$. By assumption, $\delta_{\text{env}}(t_i) \approx_{\kappa} \#i((\delta'_{\text{env}}(t_i)))$.

fn: $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \lambda t{::}\kappa_1.\, \mu :: \kappa_1 \to \kappa_2 \Rightarrow (\lambda t_{\text{env}}{::}|\Delta'_{\text{env}}|.\, \lambda t{::}\kappa_1.\, \mu')\mu_{\text{env}}$. Let $\mu_1 \approx_{\kappa_1} \mu'_1$. I must show $(\lambda t{::}\kappa_1.\, \delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu))\, \mu_1 \approx_{\kappa_2} (\delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\lambda t_{\text{env}}{::}|\Delta'_{\text{env}}|.\, \lambda t{::}\kappa_1.\, \mu')\ \mu_{\text{env}})\ \mu'_1$. Since the code of the closure is closed, it suffices to show $(\delta_{\text{env}} \uplus \delta_{\text{arg}} \uplus \{t{=}\mu_1\})(\mu) \approx_{\kappa_2} (\{t_{\text{env}}{=}\delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_{\text{env}}), t{=}\mu'_1\}(\mu'))$. Since $\Delta'_{\text{env}}; \{t{::}\kappa_1\} \vdash \mu :: \kappa_2$, I can drop the bindings of variables in $\delta_{\text{env}} \uplus \delta_{\text{arg}}$ that do not occur in Δ'_{env}. Let $\delta''_{\text{env}} = \{t{=}\delta_{\text{env}} \uplus \delta_{\text{arg}}(t) \mid t \in Dom(\Delta'_{\text{env}})\}$. I must now show $(\delta''_{\text{env}} \uplus \{t{=}\mu_1\})(\mu) \approx_{\kappa_2} (\{t_{\text{env}}{=}\delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_{\text{env}}), t{=}\mu'_1\}(\mu'))$. By the inductive hypothesis, this holds if I can show that $\delta''_{\text{env}} \uplus \{t{=}\mu_1\} \approx_{\Delta_{\text{env}};\{t::\kappa_1\}} \{t_{\text{env}}{=}\delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_{\text{env}})\} \uplus \{t{=}\mu'_1\}$. By assumption, $\mu_1 \approx_{\kappa_1} \mu'_1$ so I only need to show that

$$\delta''_{\text{env}} \approx_{\Delta_{\text{env}}} \{t_{\text{env}}{=}\delta'_{\text{env}} \uplus \delta'_{\text{arg}}(\mu_{\text{env}})\}.$$

But, this holds directly from lemma 6.4.2.

trec: We have

$$\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a) :: \kappa \Rightarrow \mathsf{Typerec}\ \mu'\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a').$$

By assumption, $\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu) \approx_\Omega \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mu')$. Therefore, these two constructors have the same normal forms when their respective substitutions are applied. Let μ_0 be this normal form. I argue by induction on the structure of μ_0 that

$$\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)) \approx_\kappa \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mathsf{Typerec}\ \mu'\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a')).$$

If μ_0 is Int, then

$$\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)) \equiv \delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_i)$$

and

$$\delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mathsf{Typerec}\ \mu'\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a')) \equiv \delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_i').$$

By the outer inductive hypothesis, these two constructors are related at κ. Similar reasoning shows that the result holds for $\mu_0 = \mathsf{Float}$ and $\mu_0 = \mathsf{Unit}$.

If μ_0 is $\mathsf{Prod}(\mu_1, \mu_2)$, then

$$\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)) \equiv \delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_p\ \mu_1\ \mu_2\ \mu_a\ \mu_b)$$

where $\mu_a = \mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)$ and $\mu_b = \mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i; \mu_f; \mu_u; \mu_p; \mu_a)$. Likewise,

$$\delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mathsf{Typerec}\ \mu'\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a')) \equiv \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mu_p'\ \mu_1\ \mu_2\ \mu_a'\ \mu_b')$$

where $\mu_a' = \mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a')$ and $\mu_b' = \mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_i'; \mu_f'; \mu_u'; \mu_p'; \mu_a')$. By the inner induction hypothesis, $\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_a) \approx_\kappa \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mu_a')$ and $\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_b) \approx_\kappa \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mu_b')$. By the outer induction hypothesis, $\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_p) \approx_{\kappa'} \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mu_p')$ where $\kappa' = \Omega \to \Omega \to \kappa \to \kappa \to \kappa$. Hence, by the definition of the relations at arrow types, we know that

$$\delta_{\text{env}} \uplus \delta_{\text{arg}}(\mu_p\ \mu_1\ \mu_2\ \mu_a\ \mu_b) \approx_\kappa \delta_{\text{env}}' \uplus \delta_{\text{arg}}'(\mu_p'\ \mu_1\ \mu_2\ \mu_a'\ \mu_b').$$

Similar reasoning shows that the result holds for $\mu_0 = \mathsf{Arrow}([\mu_1, \cdots, \mu_k], \mu)$.

□

6.4.2 Type Correctness of the Term Translation

The next step in proving the correctness of closure conversion is to show that each translated term has the translated type. I begin by showing that the type translation commutes with substitution.

Lemma 6.4.4 *If* $\Delta_{\text{env}};\Delta_{\text{arg}} \uplus \{t{::}\kappa\} \vdash \sigma \Rightarrow \sigma'$ *and* $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash \mu :: \kappa \Rightarrow \mu'$, *then* $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash \{\mu/t\}\sigma \Rightarrow \{\mu'/t\}\sigma'$.

Proof: By induction on the derivation of $\Delta_{\text{env}};\Delta_{\text{arg}} \uplus \{t{::}\kappa\} \vdash \sigma \Rightarrow \sigma'$. The base case, $T(\mu)$ relies upon the correctness of the constructor translation. □

The following lemma is critical for showing that closures are well-formed. Roughly speaking, it shows that a type obtained from the "current" constructor context $(\Delta_{\text{env}};\Delta_{\text{arg}})$ is equivalent to the type obtained from the closure's context, as long as we substitute the closure's environment for the abstracted environment variable.

Lemma 6.4.5 *If* $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash \sigma \Rightarrow \sigma_1$ *and* $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash_{\text{env}} \Delta'_{\text{env}} \Rightarrow \mu$, *then* $\Delta'_{\text{env}};\emptyset \vdash \sigma \Rightarrow \sigma_2$ *and* $\{t_{\text{env}}{::}|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}} \vdash \sigma_1 \equiv \{\mu/t_{\text{env}}\}\sigma_2$.

Theorem 6.4.6 (Type Correctness) *If* $\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\Gamma_{\text{arg}} \vdash e : \sigma \Rightarrow e'$, *then* $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash_{\text{env-type}} \Gamma_{\text{env}} \Rightarrow \Gamma'_{\text{env}}$, $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash_{\text{env-type}} \Gamma_{\text{arg}} \Rightarrow \Gamma'_{\text{arg}}$, *and* $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash \sigma \Rightarrow \sigma'$, *then* $\{t_{\text{env}}{::}|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}};\{x_{\text{env}}{:}|\Gamma'_{\text{env}}|\} \uplus \Gamma'_{\text{arg}} \vdash e' : \sigma'$.

Proof: By induction on the derivation of $\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\Gamma_{\text{arg}} \vdash e : \sigma \Rightarrow e'$. The most interesting cases are the translations of variables and λ-abstractions (shown below). The other cases follow in a straightforward fashion. In particular, the treatment of Λ-abstractions almost directly follows the treatment of λ-abstractions.

var-arg: We have $\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\{x_1{:}\sigma_1,\cdots,x_n{:}\sigma_n\} \vdash x_i : \sigma_i \Rightarrow e'$. By the **type-env** translation rule, $\Gamma'_{\text{arg}} = \{x_1{:}\sigma'_1,\cdots,x_n{:}\sigma'_n\}$ where $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash \sigma_i \Rightarrow \sigma'_i$. Thus, $\{t_{\text{env}}{::}|\Delta_{\text{env}}|\};\Delta_{\text{arg}};\{x_{\text{env}}{:}|\Gamma'_{\text{env}}|\};\Gamma'_{\text{arg}} \vdash x_i : \sigma'_i$.

var-env: We have $\Delta_{\text{env}};\Delta_{\text{arg}};\{x_1{:}\sigma_1,\cdots,x_n{:}\sigma_n\};\Gamma_{\text{arg}} \vdash x_i : \sigma_i \Rightarrow \#i(x_{\text{env}})$. By the **type-env** translation rule, $\Gamma'_{\text{env}} = \{x_1{:}\sigma'_1,\cdots,x_n{:}\sigma'_n\}$ where $\Delta_{\text{env}};\Delta_{\text{arg}} \vdash \sigma_i \Rightarrow \sigma'_i$. Thus, $|\Gamma'_{\text{env}}| = \langle\sigma'_1 \times \cdots \times \sigma'_n\rangle$. Hence, $\{t_{\text{env}}{::}|\Delta_{\text{env}}|\};\Delta_{\text{arg}};\{x_{\text{env}}{:}|\Gamma'_{\text{env}}|\};\Gamma'_{\text{arg}} \vdash \#i(x_{\text{env}}) : \sigma'_i$.

abs: To simplify the proof, I only show the case for 1-argument functions. We have

$$\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\Gamma_{\text{arg}} \vdash \lambda x{:}\sigma_a.\, e : \sigma_a \to \sigma_b \Rightarrow \langle\langle e_c, \mu_{\text{env}}, e_{\text{env}}\rangle\rangle$$

where e_c is $\texttt{vcode}[t_{\text{env}}{::}|\Delta'_{\text{env}}|, x_{\text{env}}{:}|\Gamma''_{\text{env}}|, x{:}\sigma''_a].e'$. By the inductive hypothesis, we know that

$$\{t_{\text{env}}{::}|\Delta'_{\text{env}}|\};\{x_{\text{env}}{:}|\Gamma''_{\text{env}}|\} \uplus \{x{:}\sigma''_a\} \vdash e' : \sigma''_b,$$

where $\Delta'_{\text{env}};\emptyset \vdash \Gamma'_{\text{env}} \Rightarrow \Gamma''_{\text{env}}$, $\Delta'_{\text{env}};\emptyset \vdash \sigma_a \Rightarrow \sigma''_a$, and $\Delta'_{\text{env}};\emptyset \vdash \sigma_b \Rightarrow \sigma''_b$. From this and the typing rule for `vcode`, we can conclude that

$$\emptyset;\emptyset \vdash e_c : \mathsf{code}(t_{\text{env}}{::}|\Delta'_{\text{env}}|, |\Gamma''_{\text{env}}|, \sigma''_a \to \sigma''_b).$$

From kind-preservation of the constructor translation, we know that $\{t_{\text{env}}::|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}} \vdash \mu_{\text{env}} :: |\Delta'_{\text{env}}|$. Thus, the code and the type environment agree on kinds. I only need to show that the code and the value environment agree on types.

Suppose $\Gamma'_{\text{env}} = \{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\}$. Then $e_{\text{env}} = \langle e_1, \cdots, e_n\rangle$ where

$$\Delta_{\text{env}}; \Delta_{\text{arg}}; \Gamma_{\text{env}}; \Gamma_{\text{arg}} \vdash x_i : \sigma_i \Rightarrow e_i$$

for $1 \leq i \leq n$. By the induction hypothesis,

$$\{t_{\text{env}}::|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}}; \{x_{\text{env}}{:}|\Gamma'|\} \uplus \Gamma'_{\text{arg}} \vdash e_i : \sigma'_i$$

where $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \Gamma_{\text{env}} \Rightarrow \Gamma'$, $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \Gamma_{\text{arg}} \Rightarrow \Gamma'_{\text{arg}}$, and $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma_i \Rightarrow \sigma'_i$. From lemma 6.4.5, we know that $\{t_{\text{env}}::|\Delta'_{\text{env}}|\} \vdash \sigma'_i \equiv \{\mu_{\text{env}}/t_{\text{env}}\}\Gamma''_{\text{env}}(x_i)$. Therefore,

$$\{t_{\text{env}}::|\Delta'_{\text{env}}|\} \vdash \langle \sigma'_1 \times \cdots \times \sigma'_n\rangle \equiv \{\mu_{\text{env}}/t_{\text{env}}\}|\Gamma''_{\text{env}}|.$$

Thus, from the formation rule for closures, we can conclude that

$$\{t_{\text{env}}::|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}}; \{x{:}|\Gamma'|\} \uplus \Gamma'_{\text{arg}} \vdash \langle\langle e_c, \mu_{\text{env}}, e_{\text{env}}\rangle\rangle : \{\mu_{\text{env}}/t_{\text{env}}\}(\sigma'_a \to \sigma'_b).$$

Suppose $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma_a \Rightarrow \sigma'_a$ and $\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma_b \Rightarrow \sigma'_b$. Then by the type translation,

$$\Delta_{\text{env}}; \Delta_{\text{arg}} \vdash \sigma_a \to \sigma_b \Rightarrow \sigma'_a \to \sigma'_b.$$

By lemma 6.4.5, we can conclude that

$$\{t_{\text{env}}::|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}} \vdash \sigma'_a \to \sigma'_b \equiv \{\mu_{\text{env}}/t_{\text{env}}\}(\sigma''_a \to \sigma''_b).$$

Hence,

$$\{t_{\text{env}}::|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}}; \{x{:}|\Gamma'|\} \uplus \Gamma'_{\text{arg}} \vdash \langle\langle e_c, \mu_{\text{env}}, e_{\text{env}}\rangle\rangle : \sigma'_a \to \sigma'_b.$$

□

6.4.3 Correctness of the Term Translation

Correctness of the term translation follows a similar pattern to that of the constructor translation. I begin by establishing a set of relations for closed term values, indexed by closed source types.

$$\frac{\emptyset;\emptyset \vdash e : \sigma \quad \emptyset \vdash \sigma \Rightarrow \sigma' \quad \emptyset;\emptyset \vdash e' : \sigma' \qquad e \Downarrow v \text{ iff } e' \Downarrow v' \text{ and } v \approx_\sigma v'}{e \sim_\sigma e'}$$

$$i \approx_{\mathsf{int}} i \qquad f \approx_{\mathsf{float}} f \qquad \langle\rangle \approx_{\mathsf{unit}} \langle\rangle$$

$$\frac{\pi_1\, v \sim_{\sigma_1} \pi_1\, v' \qquad \pi_2\, v \sim_{\sigma_2} \pi_1\, v'}{v \approx_{\sigma_1 \times \sigma_2} v'}$$

$$\frac{v_1 \approx_{\sigma_1} v_1' \quad \cdots \quad v_k \approx_{\sigma_k} v_k' \text{ implies } v\,[v_1, \cdots, v_k] \sim_{\sigma} v'\,[v_1', \cdots, v_k']}{v \approx_{[\sigma_1,\cdots,\sigma_k]\to\sigma} v'}$$

$$\frac{\mu \approx_{\kappa} \mu' \text{ implies } v\,[\mu] \sim_{\{\mu/t\}\sigma} v'\,[\mu']}{v \approx_{\forall t::\kappa.\sigma} v'}$$

Two expressions e and e' are related at source type σ iff e has type σ, e' has a type σ' obtained by translating σ, and e evaluates to a value iff e' evaluates to a related value at σ. Values at base type are related iff they are syntactically equal. Values at product type are related if projecting their components yields related computations. Values at arrow types are related when they yield related computations, given related arguments. Finally, two polymorphic values are related if they yield related computations given related constructors.

For open expressions, I extend the relations to substitutions γ and γ', mapping variables to values, where the relations are indexed by a source type assignment Γ as follows:

$$\frac{v_1 \approx_{\sigma_1} v_1' \quad \cdots \quad v_n \approx_{\sigma_n} v_n'}{\{x_1{=}v_1, \cdots, x_n{=}v_n\} \approx_{\{x_1:\sigma_1,\cdots,x_n:\sigma_n\}} \{x_{\mathrm{env}}{=}\langle v_1', \cdots, v_n'\rangle\}}$$

I relate pairs of substitutions, $\gamma_{\mathrm{env}};\gamma_{\mathrm{arg}}$ and $\gamma'_{\mathrm{env}};\gamma'_{\mathrm{arg}}$ as follows:

$$\frac{\gamma_{\mathrm{env}} \approx_{\Gamma_{\mathrm{env}}} \gamma'_{\mathrm{env}} \\ \mathit{Dom}(\Gamma_{\mathrm{arg}}) = \mathit{Dom}(\gamma_{\mathrm{arg}}) = \mathit{Dom}(\gamma'_{\mathrm{arg}}) \qquad \forall x \in \mathit{Dom}(\Gamma_{\mathrm{arg}}).\gamma_{\mathrm{arg}}(x) \approx_{\Gamma_{\mathrm{arg}}(x)} \gamma'_{\mathrm{arg}}(x)}{\gamma_{\mathrm{env}};\gamma_{\mathrm{arg}} \approx_{\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}}} \gamma'_{\mathrm{env}};\gamma'_{\mathrm{arg}}}$$

The following lemma shows that the translation of variables is correct, and thus so is the translation of environments.

Lemma 6.4.7 *Let* $\delta_{\mathrm{env}};\delta_{\mathrm{arg}} \approx_{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}}} \delta'_{\mathrm{env}};\delta'_{\mathrm{arg}}$ *and* $\gamma_{\mathrm{env}};\gamma_{\mathrm{arg}} \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}})} \gamma'_{\mathrm{env}};\gamma'_{\mathrm{arg}}$.

1. *If* $\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \vdash x : \sigma \Rightarrow e$, *then* $\gamma_{\mathrm{env}} \uplus \gamma_{\mathrm{arg}}(x) \sim_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\sigma)} \gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(e)$.

2. *If* $\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \vdash_{\mathrm{env}} \Gamma'_{\mathrm{env}} \Rightarrow e_{\mathrm{env}}$, *then* $\gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(e_{\mathrm{env}}) \Downarrow v_{\mathrm{env}}$ *for some* v_{env} *and,* $\{x{=}\gamma_{\mathrm{env}} \uplus \gamma_{\mathrm{arg}}(x) \mid x \in \mathit{Dom}(\Gamma'_{\mathrm{env}})\} \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\Gamma'_{\mathrm{env}})} \{x_{\mathrm{env}}{=}v_{\mathrm{env}}\}$.

With this lemma in hand, I can establish the correctness of the translation by showing that a λ_i^{ML}-Rep expression is always related to its λ_i^{ML}-Close translation, given appropriately related substitutions.

Theorem 6.4.8 (Correctness) *Let* $\delta_{\mathrm{env}};\delta_{\mathrm{arg}} \approx_{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}}} \delta'_{\mathrm{env}};\delta'_{\mathrm{arg}}$, *and let* $\gamma_{\mathrm{env}};\gamma_{\mathrm{arg}} \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}})} \gamma'_{\mathrm{env}};\gamma'_{\mathrm{arg}}$. *If* $\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \vdash e : \sigma \Rightarrow e'$, *then* $\delta_{\mathrm{env}} \uplus \delta_{\mathrm{arg}}(\gamma_{\mathrm{env}} \uplus \gamma_{\mathrm{arg}}(e)) \sim_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\sigma)} \delta'_{\mathrm{env}} \uplus \delta'_{\mathrm{arg}}(\gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(e'))$.

Proof: By induction on the derivation of $\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \vdash e : \sigma \Rightarrow e'$ (see Figure 6.3). The **var-arg** and **var-env** cases follow directly from lemma 6.4.7. The **int**, **float**, and **unit** rules follow trivially. The elimination rules **proj**, **app**, and **tapp** follow directly from the inductive hypotheses as well as the definitions of the relations. The **typerec** rule follows directly from constructor translation correctness, the inductive hypotheses, and the definition of the relations. Arguments for the **abs** and **tabs** rules follow.

abs: Let $v_1 \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\sigma_1)} v'_1, \cdots, v_k \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\sigma_k)} v'_k$, and let $\gamma''_{\mathrm{env}} = \{x{=}\gamma_{\mathrm{env}}\uplus\gamma_{\mathrm{arg}}(x) \mid x \in \mathit{Dom}(\Gamma')\}$. By lemma 6.4.7, $\gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(e_{\mathrm{env}}) \Downarrow v_{\mathrm{env}}$ for some v_{env} and $\gamma''_{\mathrm{env}} \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\Gamma'_{\mathrm{env}})} \{x_{\mathrm{env}}{=}v_{\mathrm{env}}\}$.

Let $\delta''_{\mathrm{env}} = \{t{=}\delta_{\mathrm{env}} \uplus \delta_{\mathrm{arg}}(t) \mid t \in \mathit{Dom}(\Delta')\}$ and let $\mu'_{\mathrm{env}} = \delta'_{\mathrm{env}} \uplus \delta'_{\mathrm{arg}}(\mu_{\mathrm{env}})$. By lemma 6.4.2, we know that $\delta''_{\mathrm{env}} \approx_{\Delta'_{\mathrm{env}}} \{t_{\mathrm{env}}{=}\mu'_{\mathrm{env}}\}$. Hence, $\delta''_{\mathrm{env}};\emptyset \approx_{\Delta'_{\mathrm{env}};\emptyset} \{t_{\mathrm{env}}{=}\mu'_{\mathrm{env}}\};\emptyset$. By the induction hypothesis and type preservation, we can conclude that

$$\begin{aligned}\delta''_{\mathrm{env}}(\gamma''_{\mathrm{env}} \uplus \{x_1{=}v_1, \cdots, x_k{=}v_k\}(e)) \;\approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\sigma)}\\ \{t_{\mathrm{env}}{=}\mu'_{\mathrm{env}}\}(\{x_{\mathrm{env}}{=}v_{\mathrm{env}}, x_1{=}v'_1, \cdots, x_k{=}v'_k\}(e')).\end{aligned}$$

Thus,

$$\begin{aligned}\delta_{\mathrm{env}} \uplus \delta_{\mathrm{arg}}(\gamma_{\mathrm{env}} \uplus \gamma_{\mathrm{arg}}(\lambda[x_1{:}\sigma_1, \cdots, x_k{:}\sigma_k].\, e)) \;\approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}([\sigma_1,\cdots,\sigma_k]\to\sigma)}\\ \delta'_{\mathrm{env}} \uplus \delta'_{\mathrm{arg}}(\gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(\langle\langle e_c, \mu_{\mathrm{env}}, e_{\mathrm{env}}\rangle\rangle))\end{aligned}$$

where e_c is $\mathtt{vcode}[t_{\mathrm{env}}{::}|\Delta'_{\mathrm{env}}|, x_{\mathrm{env}}{:}|\Gamma''_{\mathrm{env}}|, x_1{:}\sigma'_1, \cdots, x_k{:}\sigma'_k].e'$.

tabs: Let $\mu \approx_\kappa \mu'$ and let $\gamma''_{\mathrm{env}} = \{x{=}\gamma_{\mathrm{env}} \uplus \gamma_{\mathrm{arg}}(x) \mid x \in \mathit{Dom}(\Gamma')\}$. By lemma 6.4.7, $\gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(e_{\mathrm{env}}) \Downarrow v_{\mathrm{env}}$ for some v_{env} and $\gamma''_{\mathrm{env}} \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\Gamma'_{\mathrm{env}})} \{x_{\mathrm{env}}{=}v_{\mathrm{env}}\}$.

Let $\delta''_{\mathrm{env}} = \{t{=}\delta_{\mathrm{env}} \uplus \delta_{\mathrm{arg}}(t) \mid t \in \mathit{Dom}(\Delta')\}$ and let $\mu'_{\mathrm{env}} = \delta'_{\mathrm{env}} \uplus \delta'_{\mathrm{arg}}(\mu_{\mathrm{env}})$. By lemma 6.4.2, we know that $\delta''_{\mathrm{env}} \approx_{\Delta'_{\mathrm{env}}} \{t_{\mathrm{env}}{=}\mu'_{\mathrm{env}}\}$. Hence, by the assumption regarding μ and μ', $\delta''_{\mathrm{env}};\{t{=}\mu\} \approx_{\Delta'_{\mathrm{env}};\{t::\kappa\}} \{t_{\mathrm{env}}{=}\mu'_{\mathrm{env}}\};\{t{=}\mu'\}$. By the induction hypothesis, and type preservation, we can conclude that

$$\delta''_{\mathrm{env}} \uplus \{t{=}\mu\}(\gamma''_{\mathrm{env}}(e)) \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}\uplus\{t=\mu\}(\sigma)} \{t_{\mathrm{env}}{=}\mu'_{\mathrm{env}}, t{=}\mu'\}(\{x_{\mathrm{env}}{=}v_{\mathrm{env}}\}(e')).$$

Thus,

$$\delta_{\mathrm{env}} \uplus \delta_{\mathrm{arg}}(\gamma_{\mathrm{env}} \uplus \gamma_{\mathrm{arg}}(\Lambda t{::}\kappa.e)) \approx_{\delta_{\mathrm{env}}\uplus\delta_{\mathrm{arg}}(\forall t::\kappa.\sigma)} \delta'_{\mathrm{env}} \uplus \delta'_{\mathrm{arg}}(\gamma'_{\mathrm{env}} \uplus \gamma'_{\mathrm{arg}}(\langle\langle e_c, \mu_{\mathrm{env}}, e_{\mathrm{env}}\rangle\rangle))$$

where e_c is $\mathtt{tcode}[t_{\mathrm{env}}{::}|\Delta'_{\mathrm{env}}|, x_{\mathrm{env}}{:}|\Gamma''_{\mathrm{env}}|, t{::}\kappa].e'$. □

6.5 Related Work

Closure conversion is discussed in descriptions of various functional language compilers [111, 78, 11, 9, 109]. It is closely related to λ-lifting [69] in that it eliminates free variables in the bodies of λ-abstractions. However, closure conversion differs by making the representation of the environment explicit as a data structure. Making the environment explicit is important because it exposes environment construction and variable lookup to an optimizer. Furthermore, Shao and Appel show that not all environment representations are "safe for space" [109], and thus choosing a good environment representation is an important part of compilation.

Wand and Steckler [124] have considered two optimizations of the basic closure conversion strategy — *selective* and *lightweight* closure conversion — and provide a correctness proof for each of these in an untyped setting. Hannan [54] recasts Wand's work into a typed setting, and provides correctness proofs for Wand's optimizations. As with my translation, Hannan's translation is formulated as a deductive system. However, Hannan does not consider the important issue of environment representation (preferring an abstract account), nor does he consider the typing properties of the closure-converted code.

Minamide, Morrisett, and Harper give a comprehensive treatment of type-directed closure conversion for the simply-typed λ-calculus and a predicative, type-passing polymorphic λ-calculus [92, 91]. This chapter extends the initial treatment by showing how to closure convert a language like λ_i^{ML} with higher-kinds (i.e., functions at both the constructor and term levels).

Chapter 7

Types and Garbage Collection

In the previous chapters, I argued that one should use types at *compile* time to direct the translation of a high-level language to a low-level language. In this chapter, I will show that types can be used at *run* time to implement a key facility, namely automatic storage reclamation or *garbage collection*. As in the previous chapters, types will guide us in the process of garbage collection as well as a proof of correctness.

In most accounts of language implementation, garbage collection is either ignored or at best discussed without regard to the rest of the implementation. Most descriptions of garbage collectors are extremely low-level and concentrate on manipulating "mark bits", "forwarding pointers", "tags", "reference counts", and the like. This focus on the low-level details makes it extremely difficult to determine what effect a garbage collector has on a program's evaluation. As a result, there are very few proofs that a garbage collector does not interfere with evaluation and only collects true garbage.

The primary culprit is that traditional models of evaluation based on the λ-calculus, such as the contextual semantics of Mini-ML and λ_i^{ML}, use *substitution* as the mechanism of computation. Unfortunately, substitution hides all memory management issues: during evaluation we simply α-convert terms so that we can always find an unused variable. Since α-conversion is defined in terms of substitution, it is substitution that implicitly "allocates" fresh variable names for us. Furthermore, when we substitute a value for a variable, if there are no occurrences of that variable, the value disappears. Thus, substitution also takes care of "collecting" unneeded terms.

In this chapter, I develop an alternative style of semantics where allocation is explicit. The basic idea is to represent a program's memory or *heap* as a global set of syntactic declarations. The evaluation rules allocate large objects in the global heap and automatically dereference pointers to such objects when needed. Since the heap is explicit, the process of garbage collection is made explicit as any relation that removes portions of a program's heap without affecting the program's evaluation.

I specify a particular garbage collection strategy which characterizes the family of *tracing* garbage collectors including mark/sweep and copying collectors. By employing standard syntactic techniques, I prove that the specification is correct with respect to the definition of garbage.

Next, I develop an algorithm that implements the tracing garbage collection strategy. Standard tracing collectors use *tags* on values in the heap to determine their *shape* — the size of the object and any pointers contained in the object. Instead of using tags, I show that for monomorphic languages, if enough type information is recorded on terms at compile time, types can be used to determine the shape of objects in the heap. Consequently, no tags are required on the values in the heap to support garbage collection. This approach to *tag-free* garbage collection is not new [23], but my formulation is at a sufficiently high level that it is easy to prove its correctness.

I then show how to extend the tag-free collection algorithm to accommodate *generational* garbage collection. Generational collection is an important technique that collects most of the garbage in a program but examines a smaller set of objects than the standard tracing collection algorithm. Thus, generational collection tends to improve the latency or response time of garbage collection without sacrificing too much space.

After showing how a monomorphic language can be garbage collected in a tag-free fashion, I show how a type-passing, polymorphic language such as λ_i^{ML} can be garbage collected. The key idea is to use constructors that are passed dynamically as arguments to procedures during the garbage collection process. With type information recorded at compile time, this allows us to reconstruct the shape of all objects. Hence, tag-free garbage collection is another mechanism that can use dynamic type dispatch to account for variable types. As for monomorphic languages, this approach to tag-free garbage collection for polymorphic languages is not new [119, 6, 96, 95], but my formulation is sufficiently abstract that we can easily prove its correctness.

Tag-free garbage collection is important for two very practical reasons: first, a clever tag-free implementation can avoid manipulating any type information in monomorphic code at run time, except during garbage collection. In contrast, a tagging implementation must tag values as they are created and possibly untag the values when they are examined. The overheads of manipulating these tags during the computation can be considerable [112] and implementors go to great lengths and use many clever encodings to minimize these overheads [128]. Second, tag-free garbage collection supports language and system *interoperability*. In particular, many ubiquitous languages, such as Fortran, C, and C++, do *not* provide automatic memory management and, thus, do not tag values. A language that uses tags for collection must strip tags off values before passing them to library routines written in Fortran or C. Similarly, communicating with the operating system, windowing system, or hardware requires matching the representations dictated by these systems. Since tag-free collection places no constraints on the representation of values, communicating with these systems is easier and more efficient.

(types)	τ	$::=$	$\mathsf{int} \mid \mathsf{float} \mid \mathsf{unit} \mid \langle \tau_1 \times \tau_2 \rangle \mid \mathsf{code}(\tau_1, \tau_2) \mid \tau_1 \to \tau_2$
(expressions)	e	$::=$	$\mathtt{return}\ x \mid \mathtt{if0}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{let}\ x{:}\tau = d\ \mathtt{in}\ e$
(declarations)	d	$::=$	$x \mid i \mid f \mid \langle\rangle \mid \langle x_1, x_2 \rangle \mid \pi_1\, x \mid \pi_2\, x \mid$ $\mathtt{vcode}[x_{\mathrm{env}}{:}\tau_{\mathrm{env}}, x{:}\tau].e \mid \langle\langle x_1, x_2 \rangle\rangle \mid x\, x' \mid$ $\mathtt{eqint}(x_1, x_2) \mid \mathtt{eqfloat}(x_1, x_2)$

Figure 7.1: Syntax of Mono-GC Expressions

7.1 Mono-GC

In this Section, I define a language called Mono-GC, which is derived from the monomorphic subset of the λ_i^{ML}-Close language (see Chapter 6). The expressions of the language are limited in the style of Flanagan et al.[42]. In particular, the language forbids nested expressions and requires that all values and computations be bound to variables. These restrictions simplify the presentation of the semantics, but they provide many of the practical benefits of CPS [9].

The syntax of Mono-GC is defined in Figure 7.1. Types are monomorphic and include base types, products, code, and arrow types. To simplify the language, I only consider functions of one argument. Expressions (e) return a variable, branch on a variable, or bind a declaration (d) to a variable and continue with some expression. Declarations can either be immediate constants, a primitive operation (e.g., `eqint`) applied to some variables, a tuple whose components are variables, a piece of code, a closure, a projection from a variable, or an application of a variable to a variable. As in Mono-CLOSE, I require that code always be closed.

The declaration $\mathtt{let}\ x{:}\tau = d\ \mathtt{in}\ e$ binds the variable x within the scope of the expression e. Similarly, the declaration $\mathtt{vcode}[x_{\mathrm{env}}{:}\tau_{\mathrm{env}}, x{:}\tau].e$ binds the variables x_{env} and x in the scope of the expression e. I consider expressions to be equivalent up to α-conversion of the bound variables.

From an implementor's perspective, the variables of Mono-GC correspond to abstract machine registers. In contrast to a semantics based on substitution, I bind values to registers and then compute with the registers, instead of substituting the contents of a register within a term. By restricting declarations so that there are no nested expressions, I greatly simplify the process of breaking an expression into an evaluation context and instruction. Indeed, the evaluation contexts are explicitly tracked via a program stack (see below).

The other syntactic classes of Mono-GC are given in Figure 7.2. Mono-GC programs have four components: a heap (H), a stack (S), a typed environment (ρ), and an expression (e). Informally, the heap holds values too large to fit into registers. The stack holds a list of delayed computations, essentially as closures, that are waiting for a function invocation to return. The environment serves as the "registers" of the abstract machine and maps variables to small values. Finally, the expression corresponds to the code that the machine is currently executing.

Formally, a heap is an *unordered* set of bindings that maps locations (l) to *heap values*. A heap value (h) is a tuple of small values, a piece of code, or a closure. Heap values are too large to fit into registers and are thus bound to locations in memory. Small values (v) are values that can fit into registers and consist of integer and floating point constants, unit, and locations. A typed environment (ρ) is an environment that maps variables to both a type and a small value. I use $\rho(x)$ and $\rho_{\text{type}}(x)$ to denote the value and type to which ρ maps x, respectively. A stack (S) is a list of pairs of the form $[\rho, \lambda x{:}\tau.\, e]$. Each pair represents a delayed computation where ρ is the environment of the computation and $\lambda x{:}\tau.\, e$ is the "continuation" of the delayed computation. I require that all the free variables in $\lambda x{:}\tau.\, e$ be bound in the environment ρ. The stack could also be represented as a list of closures, but this approach models a system where stack frames are not allocated in the heap. Composing the "closures" of the stack results in the "continuation" of the program. Finally, I distinguish programs with an empty stack and `return` x expression as *answer* programs.

Like the expression level, I consider code-expressions to bind their variable arguments within the scope of their expression components. For a stack frame $[\rho, \lambda x{:}\tau.\, e]$, I consider the domain of ρ and x to be bound within e. Finally, I consider the domain of the heap to bind the locations within the scope of the range of the heap, the stack, and the environment of a program. Thus, I consider programs to be equivalent up to α-conversion and reordering of the locations bound in the heap.

Considering programs equivalent modulo α-conversion and the treatment of heaps and environments as sets instead of sequences hides many of the complexities of memory management. In particular, programs are automatically considered equivalent if the heap or environment is rearranged and locations or variables are renamed as long as the "graph" of the program is preserved. This abstraction allows us to focus on the issues of determining what bindings in the heap are garbage without specifying how such bindings are represented in a real machine.

7.1.1 Dynamic Semantics of Mono-GC

Figure 7.3 defines the rewriting rules for Mono-GC programs. I briefly describe each rule below:

(locations)	l		
(small values)	v	$::=$	$i \mid f \mid l \mid \langle\rangle$
(heap values)	h	$::=$	$\langle v_1, v_2\rangle \mid \texttt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau].e \mid \langle\langle v_{\text{code}}, v_{\text{env}}\rangle\rangle$
(heaps)	H	$::=$	$\{l_1{=}h_1, \cdots, l_n{=}h_n\}$
(environments)	ρ	$::=$	$\{x_1{:}\tau_1{=}v_1, \cdots, x_n{:}\tau_n{=}v_n\}$
(stacks)	S	$::=$	$[] \mid S[\rho, \lambda x{:}\tau.\, e]$
(programs)	P	$::=$	(H, S, ρ, e)
(answers)	A	$::=$	$(H, [], \rho, \texttt{return}\ x)$

Figure 7.2: Syntax of Mono-GC Programs

(1,2) An `if0` is applied to some variable x. We find the value of x in the current environment and select the appropriate expression according to whether this value is 0 or some other integer.

(3) A variable x is bound to another variable x' in a `let` expression. We lookup the small value v to which x' is bound in the current environment, ρ. We extend ρ to map x to v and continue with the body of the `let`.

(4,5) The integer equality primitive is applied to two variables. We find the value of the variables in the environment and return 1 or 0 as appropriate. This new value is bound to the `let`-bound variable in the new environment. The rules for floating-point equality (not shown) are similar.

(6) The expression binds a variable to an immediate small value. We add that binding to the current environment and continue.

(7,8,9) The expression allocates a tuple, code, or a closure binding the result to x. We first replace all of the free variables in the object with their bindings from the environment. Code objects are always closed, so they are not effected. Then, we allocate a new location on the heap l and bind the heap value to this location. Next, we map x to l in the current environment and continue.

(10) The expression projects the i^{th} component of y, binding the result to x. We lookup y in the environment, find that it is bound to the location l. We dereference l in the heap and find that it is bound to a heap value $\langle v_1, v_2 \rangle$. We bind v_i to x in the environment and continue.

(11) The expression applies x_1 to some argument x_2, binding the result to x. We lookup x_1 and x_2 in the environment, finding that x_1 is bound to some location l and x_2 is bound to v. We dereference l in the heap and find that it is bound to a closure, $\langle\langle l_{\text{code}}, v_{\text{env}} \rangle\rangle$, where l_{code} is bound to the code $\texttt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, y{:}\tau'].e$. We form a new stack frame by pairing the current environment (ρ') and the current expression, abstracting the result of the function $\lambda x{:}\tau.\, e'$. This frame is pushed on the stack. Then, we install the environment which maps x_{env} to v_{env} and y to v. We then continue with the body of the closure as the current expression.

(12) The current expression is $\texttt{return}\ x'$ and the stack is non-empty. We lookup the value of x' in the current environment (ρ'), pop off a stack frame, install its environment (ρ) as the current environment, bind the small value $\rho'(x')$ to the argument of the continuation (x), and continue with the body of the continuation.

In this formulation, each time a `let`-expression is evaluated, the type ascribed to the bound variable is entered into the current environment ρ, as well as the value. In essence, the environment contains a type assignment Γ that is constructed on the fly. It is possible to avoid constructing these type assignments at run time by labelling `let`-expressions not only with the type of the bound variable, but also with the types of all variables in scope. This allows evaluation to simply discard the current type assignment and proceed with the assignment labelling the expression. Since these assignments can be calculated at compile time, no assignment construction need occur at run time.

Of course, labelling each expression with the types of all variables in scope could take a great deal of space. But, as I will show, this type information is only used during garbage collection. Most language implementations restrict garbage collection from occuring except at certain points during evaluation. For example, the garbage collector of SML/NJ is only invoked at the point when a function is called, or at the point when an array or vector is allocated. Hence, we only need to record type assignments for those `let`-expressions that perform a function call or array allocation. This guarantees that when we invoke garbage collection, enough type information is present to do the job.

7.1.2 Static Semantics of Mono-GC

The static semantics of Mono-GC is described via a family of judgments. The first two judgments, $\Gamma \vdash_{\text{exp}} e : \tau$ and $\Gamma \vdash_{\text{dec}} d : \tau$, give types to expressions and declarations,

1. $(H, S, \rho, \texttt{if0}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \longmapsto (H, S, \rho, e_1) \qquad (\rho(x) = 0)$

2. $(H, S, \rho, \texttt{if0}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \longmapsto (H, S, \rho, e_2)(\rho(x) = i \text{ and } i \neq 0)$

3. $(H, S, \rho, \texttt{let}\ x{:}\tau = x'\ \texttt{in}\ e) \longmapsto (H, S, \rho \uplus \{x{:}\tau{=}\rho(x')\}, e)$

4. $(H, S, \rho, \texttt{let}\ x{:}\tau = \texttt{eqint}(x_1, x_2)\ \texttt{in}\ e) \longmapsto$
$(H, S, \rho \uplus \{x{:}\tau{=}1\}, e) \qquad (\rho(x_1) =_{\text{int}} \rho(x_2))$

5. $(H, S, \rho, \texttt{let}\ x{:}\tau = \texttt{eqint}(x_1, x_2)\ \texttt{in}\ e) \longmapsto$
$(H, S, \rho \uplus \{x{:}\tau{=}0\}, e) \qquad (\rho(x_1) \neq_{\text{int}} \rho(x_2))$

6. $(H, S, \rho, \texttt{let}\ x{:}\tau = v\ \texttt{in}\ e) \longmapsto (H, S, \rho \uplus \{x{:}\tau{=}v\}, e)$

7. $(H, S, \rho, \texttt{let}\ x{:}\tau = \langle x_1, x_2\rangle\ \texttt{in}\ e) \longmapsto$
$(H \uplus \{l = \langle \rho(x_1), \rho(x_2)\rangle\}, S, \rho \uplus \{x{:}\tau{=}l\}, e)$

8. $(H, S, \rho, \texttt{let}\ x{:}\tau = \texttt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau'].e'\ \texttt{in}\ e) \longmapsto$
$(H \uplus \{l = \texttt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau'].e'\}, S, \rho \uplus \{x{:}\tau{=}l\}, e)$

9. $(H, S, \rho, \texttt{let}\ x{:}\tau = \langle\langle x_{\text{code}}, x_{\text{env}}\rangle\rangle\ \texttt{in}\ e) \longmapsto$
$(H \uplus \{l = \langle\langle \rho(x_{\text{code}}), \rho(x_{\text{env}})\rangle\rangle\}, S, \rho \uplus \{x{:}\tau{=}l\}, e)$

10. $(H, S, \rho, \texttt{let}\ x{:}\tau = \pi_i\, y\ \texttt{in}\ e) \longmapsto (H, S, \rho \uplus \{x{:}\tau{=}v_i\}, e)$
where $\rho(y) = l$ and $H(l) = \langle v_1, v_2\rangle \qquad (1 \leq i \leq 2)$

11. $(H, S, \rho, \texttt{let}\ x{:}\tau = x_1\, x_2\ \texttt{in}\ e') \longmapsto$
$(H, S[\rho, \lambda x{:}\tau.\, e'], \{x_{\text{env}}{:}\tau_{\text{env}}{=}v_{\text{env}}, y{:}\tau'{=}\rho(x_2)\}, e)$
where $\rho(x_1) = l$ and $H(l) = \langle\langle l_{\text{code}}, v_{\text{env}}\rangle\rangle$ and
$H(l_{\text{code}}) = \texttt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, y{:}\tau'].e$

12. $(H, S[\rho, \lambda x{:}\tau.\, e], \rho', \texttt{return}\ x') \longmapsto (H, S, \rho \uplus \{x{:}\tau{=}\rho'(x')\}, e)$

Figure 7.3: Rewriting Rules for Mono-GC

respectively, in the context of a variable type assignment Γ. These judgments are derived via the conventional axioms and inference rules of Figure 7.4, ignoring the equality primitives.

The static semantics for Mono-GC programs requires six more judgments that are characterized as follows. I use Ψ to range over location type assignments which map locations to types.

$\Psi \vdash_{\text{val}} v : \tau$	v is a well-formed small value of type τ
$\Psi \vdash_{\text{hval}} h : \tau$	h is a well-formed heap value of type τ
$\Psi \vdash_{\text{heap}} H : \Psi$	H is a well-formed heap described by Ψ
$\Psi \vdash_{\text{env}} \rho : \Gamma$	ρ is a well-formed environment described by Γ
$\Psi \vdash_{\text{stack}} S : \tau_1 \to \tau_2$	S is a well-formed stack of type $\tau_1 \to \tau_2$
$\vdash_{\text{prog}} P : \tau$	P is a well-formed program of type τ

These judgments are defined by the inference rules and axioms of Figure 7.5. A program has type τ if the following requirements are met: The program's heap can be described by Ψ under no assumptions and is thus closed. The program's stack maps τ' to τ under the assumptions of Ψ. The program's environment is described by Γ under the assumptions of Ψ. Finally, the program's expression has the type τ' under the assumptions of Γ.

A heap H is described by Ψ under the assumptions Ψ' if for all locations l in Ψ, $H(l)$ has the type $\Psi(l)$ under the assumptions of both Ψ and Ψ'. This circularity in the definition allows cycles in the heap, in much the same way that a typing rule for `fix` allows a circular definition of a recursive function. A stack has type $\tau_1 \to \tau_2$ if the composition of its closure components yields a function from τ_1 values to τ_2 values. A closure has type $\tau' \to \tau$ if its environment has type τ_{env} and its code has type $\mathsf{code}(\tau_{\text{env}}, \tau' \to \tau)$.

Lemma 7.1.1 (Extension) *If $(H, S, \rho, e) \longmapsto (H', S', \rho', e')$, then:*

1. *$H' = H \uplus H''$ for some H''*
2. *if $S = S'$, then $\rho' = \rho \uplus \rho''$ for some ρ''.*

Theorem 7.1.2 (Preservation) *If $\vdash_{\text{prog}} P : \tau$ and $P \longmapsto P'$, then $\vdash_{\text{prog}} P' : \tau$.*

Theorem 7.1.3 (Canonical Forms) *Suppose $\vdash_{\text{prog}} (H, [], \rho, \texttt{return } x) : \tau$. Then if τ is:*

- int, *then $\rho(x) = i$ for some i.*
- float, *then $\rho(x) = f$ for some f.*
- unit, *then $\rho(x) = \langle\rangle$.*

Expressions:

$$(\textbf{var})\ \Gamma \uplus \{x{:}\tau\} \vdash_{\text{var}} x : \tau \qquad (\textbf{var-e})\ \frac{\Gamma \vdash_{\text{var}} x : \tau}{\Gamma \vdash_{\text{exp}} \texttt{return}\ x : \tau}$$

$$(\textbf{if0-e})\ \frac{\Gamma \vdash_{\text{var}} x : \mathsf{int} \qquad \Gamma \vdash_{\text{exp}} e_1 : \tau \qquad \Gamma \vdash_{\text{exp}} e_2 : \tau}{\Gamma \vdash_{\text{exp}} \texttt{if0}\ x\ \texttt{then}\ e_1\ \texttt{else}\, e_2 : \tau}$$

$$(\textbf{let-e})\ \frac{\Gamma \vdash_{\text{dec}} d : \tau' \qquad \Gamma \uplus \{x{:}\tau'\} \vdash_{\text{exp}} e : \tau}{\Gamma \vdash_{\text{exp}} \texttt{let}\, x{:}\tau' = d\ \texttt{in}\ e : \tau}$$

Declarations:

$$(\textbf{var-d})\ \Gamma \uplus \{x{:}\tau\} \vdash_{\text{dec}} x : \tau \qquad (\textbf{int-d})\ \Gamma \vdash_{\text{dec}} i : \mathsf{int}$$

$$(\textbf{float-d})\ \Gamma \vdash_{\text{dec}} f : \mathsf{float} \qquad (\textbf{unit-d})\ \Gamma \vdash_{\text{dec}} \langle\rangle : \mathsf{unit}$$

$$(\textbf{tuple-d})\ \frac{\Gamma \vdash_{\text{var}} x_1 : \tau_1 \qquad \Gamma \vdash_{\text{var}} x_2 : \tau_2}{\Gamma \vdash_{\text{dec}} \langle x_2, x_2\rangle : \langle \tau_1 \times \tau_2\rangle} \qquad (\textbf{proj-d})\ \frac{\Gamma \vdash_{\text{var}} x : \langle \tau_1 \times \tau_2\rangle}{\Gamma \vdash_{\text{dec}} \pi_i\, x : \tau_i} \quad (1 \le i \le 2)$$

$$(\textbf{vcode-d})\ \frac{\{x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau'\} \vdash_{\text{exp}} e : \tau}{\Gamma \vdash_{\text{dec}} \mathtt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau'].e : \mathsf{code}(\tau_{\text{env}}, \tau' \to \tau)}$$

$$(\textbf{close-d})\ \frac{\Gamma \vdash_{\text{var}} x_{\text{code}} : \mathsf{code}(\tau_{\text{env}}, \tau) \qquad \Gamma \vdash_{\text{var}} x_{\text{env}} : \tau_{\text{env}}}{\Gamma \vdash_{\text{dec}} \langle\langle x_{\text{code}}, x_{\text{env}}\rangle\rangle : \tau}$$

$$(\textbf{app-d})\ \frac{\Gamma \vdash_{\text{var}} x_1 : \tau' \to \tau \qquad \Gamma \vdash_{\text{var}} x_2 : \tau'}{\Gamma \vdash_{\text{dec}} x_1\, x_2 : \tau}$$

Figure 7.4: Static Semantics of Mono-GC Expressions

Values:

$$(\textbf{loc-v})\ \ \Psi \uplus \{l{:}\tau\} \vdash_{\text{val}} l : \tau \qquad\qquad (\textbf{int-v})\ \ \Psi \vdash_{\text{val}} i : \mathsf{int}$$

$$(\textbf{float-v})\ \ \Psi \vdash_{\text{val}} f : \mathsf{float} \qquad\qquad (\textbf{unit-v})\ \ \Psi \vdash_{\text{val}} \langle\rangle : \mathsf{unit}$$

Heap Values:

$$(\textbf{tuple-h})\ \ \frac{\Psi \vdash_{\text{val}} v_1 : \tau_1 \quad \Psi \vdash_{\text{val}} v_2 : \tau_2}{\Psi \vdash_{\text{hval}} \langle v_1, v_2\rangle : \langle \tau_1 \times \tau_2 \rangle}$$

$$(\textbf{code-h})\ \ \frac{\vdash_{\text{dec}} \mathtt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau'].e : \mathsf{code}(\tau_{\text{env}}, \tau' \to \tau)}{\Psi \vdash_{\text{hval}} \mathtt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau'].e : \mathsf{code}(\tau_{\text{env}}, \tau' \to \tau)}$$

$$(\textbf{close-h})\ \ \frac{\Psi \vdash_{\text{val}} v_{\text{code}} : \mathsf{code}(\tau_{\text{env}}, \tau) \qquad \Psi \vdash_{\text{val}} v_{\text{env}} : \tau_{\text{env}}}{\Psi \vdash_{\text{hval}} \langle\langle v_{\text{code}}, v_{\text{env}}\rangle\rangle : \tau}$$

Heaps:

$$(\textbf{heap})\ \ \frac{\forall l \in \mathit{Dom}(\Psi). \Psi' \uplus \Psi \vdash_{\text{hval}} H(l) : \Psi(l)}{\Psi' \vdash_{\text{heap}} H : \Psi}$$

Environments:

$$(\textbf{env})\ \ \frac{\Psi \vdash_{\text{val}} v_1 : \tau_1 \quad \cdots \quad \Psi \vdash_{\text{val}} v_n : \tau_n}{\Psi \vdash_{\text{env}} \{x_1{:}\tau_1{=}v_1, \cdots, x_n{:}\tau_n{=}v_n\} : \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\}} \quad (x_1, \cdots, x_n \text{ unique})$$

Stacks:

$$(\textbf{empty-stack})\ \ \Psi \vdash_{\text{stack}} [] : \tau \to \tau$$

$$(\textbf{push-stack})\ \ \frac{\Psi \vdash_{\text{stack}} S : \tau_2 \to \tau_3 \qquad \Psi \vdash_{\text{env}} \rho : \Gamma \qquad \Gamma \uplus \{x{:}\tau_1\} \vdash e : \tau_2}{\Psi \vdash_{\text{stack}} S[\rho, \lambda x{:}\tau_1.\, e] : \tau_1 \to \tau_3}$$

Programs:

$$(\textbf{prog})\ \ \frac{\begin{array}{c}\emptyset \vdash_{\text{heap}} H : \Psi \qquad \Psi \vdash S : \tau' \to \tau \\ \Psi \vdash_{\text{env}} \rho : \Gamma \qquad \Gamma \vdash e : \tau'\end{array}}{\vdash_{\text{prog}} (H, S, \rho, e) : \tau}$$

Figure 7.5: Static Semantics of Mono-GC Programs

- $\langle\tau_1 \times \tau_2\rangle$, *then* $\rho(x) = l$ *and* $H(l) = \langle v_1, v_2\rangle$ *for some* l, v_1, *and* v_2.
- $\mathsf{code}(\tau_{\mathrm{env}}, \tau_1 \to \tau_2)$, *then* $\rho(x) = l$ *and* $H(l) = \mathtt{vcode}[x_{\mathrm{env}}{:}\tau_{\mathrm{env}}, x{:}\tau_1].e$ *for some* l, x_{env}, x, *and* e.
- $\tau_1 \to \tau_2$, *then* $\rho(x) = l$, $H(l) = \langle\langle l_{\mathrm{code}}, l_{\mathrm{env}}\rangle\rangle$, *and*

  $$H(l_{\mathrm{code}}) = \mathtt{vcode}[x_{\mathrm{env}}{:}\tau_{\mathrm{env}}, x{:}\tau_1].e,$$

  for some l, l_{code}, l_{env}, x_{env}, τ_{env}, x, *and* e.

Theorem 7.1.4 (Progress) *If* $\vdash P : \tau$, *then either* $P = A$ *for some answer* A *or else there exists some* P' *such that* $P \longmapsto P'$.

7.2 Abstract Garbage Collection

Since the semantics of Mono-GC makes the allocation of values explicit, I can define what it means to "garbage collect" a value in the heap. A binding $l = h$ in the heap of a program is garbage if removing the binding produces an "equivalent" program. To simplify the presentation, I will focus on programs that return an integer (i.e., are of type int) and use Kleene equivalence to compare programs.

Definition 7.2.1 (Kleene Equivalence) $P_1 \simeq P_1$ *means* $P_1 \Downarrow (H_1, [], \rho_1, \mathtt{return}\ x_1)$ *iff* $P_2 \Downarrow (H_2, [], \rho_2, \mathtt{return}\ x_2)$ *and* $\rho_1(x_1) = \rho_2(x_2) = i$ *for some integer* i.

Definition 7.2.2 (Heap Garbage) *If* $P = (H \uplus \{l{=}h\}, S, \rho, e)$, *and* $\vdash_{\mathtt{prog}} P : \mathsf{int}$, *then the binding* $l{=}h$ *is garbage in* P *iff* $P \simeq (H, S, \rho, e)$ *and* $\vdash_{\mathtt{prog}} (H, S, \rho, e) : \mathsf{int}$.

A *collection* of a well-typed program P is obtained by dropping a (possibly empty) set of garbage bindings from the heap of P, resulting in a well-typed program P'. This definition is very weak in that it only allows us to drop bindings in the heap and precludes other program transformations including modifications to the stack, environment, or current expression. It even precludes changing a heap value to some other heap value. A *garbage collector* is a rewriting rule that computes a collection of a program.

Many garbage collectors attempt to collect more garbage than is allowed by this definition by modifying the stack, the environment, or values in the heap in some simple manner. For example, many collectors drop unneeded bindings in the current environment or environments on the stack. This technique is known as "black-holing." Some very few collectors reclaim bindings by remapping locations from one heap value to an already existing, equivalent heap value. This is known as *hash-consing* in the garbage collection literature.

However, the definition I have given accurately models what conventional garbage collectors try to do. In fact, as I will show, the family of *tracing* garbage collectors, typified by mark/sweep and copying collectors, have the nice property that they always collect as much as this definition of garbage allows, but no more[1]. Consequently, tracing collectors are *optimal* with respect to this definition of garbage.

Abstractly, tracing collectors simply drop bindings in the heap that are not "reachable" from the stack or the current environment. I define reachability in terms of the free locations of program components, denoted $FL(-)$:

$$\begin{array}{rcl}
FL_{\mathrm{val}}(l) & = & \{l\} \\
FL_{\mathrm{val}}(v) & = & \emptyset \qquad (v = i,\ f,\ \text{or } \langle\rangle) \\
\\
FL_{\mathrm{hval}}(\langle v_1, v_2\rangle) & = & FL_{\mathrm{val}}(v_1) \cup FL_{\mathrm{val}}(v_2) \\
FL_{\mathrm{hval}}(\mathtt{vcode}[x_{\mathrm{env}}{:}\tau_{\mathrm{env}}, x{:}\tau].e) & = & \emptyset \\
FL_{\mathrm{hval}}(\langle\langle v_{\mathrm{code}}, v_{\mathrm{env}}\rangle\rangle) & = & FL_{\mathrm{val}}(v_{\mathrm{code}}) \cup FL_{\mathrm{val}}(v_{\mathrm{env}}) \\
\\
FL_{\mathrm{heap}}(\{l_1 = h_1, \cdots, l_n = h_n\}) & = & (\cup_{i=1}^{n}(FL_{\mathrm{hval}}(h_i))) \setminus \{l_1, \cdots, l_n\} \\
\\
FL_{\mathrm{env}}(\{x_1{=}v_1, \cdots, x_n{=}v_n\}) & = & \cup_{i=1}^{n} FL_{\mathrm{val}}(v_i) \\
\\
FL_{\mathrm{stack}}([]) & = & \emptyset \\
FL_{\mathrm{stack}}(S[\rho, \lambda x{:}\tau.\, e]) & = & FL_{\mathrm{stack}}(S) \cup FL_{\mathrm{env}}(\rho) \\
\\
FL_{\mathrm{prog}}(H, S, \rho, e) & = & (FL_{\mathrm{heap}}(H) \cup FL_{\mathrm{stack}}(S) \cup FL_{\mathrm{env}}(\rho)) \setminus Dom(H)
\end{array}$$

A tracing collector is any collector that drops bindings in the heap but does not leave any free locations. I represent this specification as a new rewriting rule as follows:

Definition 7.2.3 (Tracing Collector) $(H \uplus H', S, \rho, e) \stackrel{\mathrm{trace}}{\longmapsto} (H, S, \rho, e)$ *if and only if* $FL_{\mathrm{prog}}(H, S, \rho, e) = \emptyset$.

I must show that tracing garbage collection is indeed a garbage collector in that, when given a program, it always produces a Kleene-equivalent program. The keys to a simple, syntactic proof of correctness are *Postponement* and *Diamond* Lemmas. The statements of these lemmas can be summarized by the diagrams of Figure 7.6, respectively, where solid arrows denote relations that are assumed to exist and dashed arrows denote relations that can be derived from the assumed relations.

[1]In an untyped setting where collections are not required to be closed programs, it is undecidable whether or not a given binding in an arbitrary program is garbage [96, 95]. This more general notion of garbage can be recovered in the typed setting by allowing locations to be rebound in the heap.

$$
\begin{tikzcd}
P_1 \arrow[r, "\text{trace}"] \arrow[d, dashed, "\text{R}"'] & P_2 \arrow[d, "\text{R}"] \\
P_2' \arrow[r, dashed, "\text{trace}"'] & P_3
\end{tikzcd}
\qquad
\begin{tikzcd}
P_1 \arrow[r, "\text{trace}"] \arrow[d, "\text{R}"'] & P_2 \arrow[d, dashed, "\text{R}"] \\
P_2' \arrow[r, dashed, "\text{trace}"'] & P_3
\end{tikzcd}
$$

Figure 7.6: Postponement and Diamond Properties

Lemma 7.2.4 (Postponement) *If* $P_1 \overset{\text{trace}}{\longmapsto} P_2 \longmapsto P_3$*, then there exists a* P_2' *such that* $P_1 \longmapsto P_2' \overset{\text{trace}}{\longmapsto} P_3$.

Lemma 7.2.5 (Diamond) *If* $P_1 \overset{\text{trace}}{\longmapsto} P_2$ *and* $P_1 \longmapsto P_2'$*, then there exists a* P_3 *such that* $P_2 \longmapsto P_3$ *and* $P_2' \overset{\text{trace}}{\longmapsto} P_3$.

With the Postponement and Diamond Lemmas, it is straightforward to show that tracing garbage collection does not affect evaluation.

Theorem 7.2.6 (Correctness of Tracing Collection) *If* $P \overset{\text{trace}}{\longmapsto} P'$*, then* P' *is a collection of* P.

Proof: Let $P = (H_1 \uplus H_2, S, \rho, e)$ and let $P' = (H_1, S, \rho, e)$ such that $P \overset{\text{trace}}{\longmapsto} P'$. I must show P evaluates to an integer answer iff P' evaluates to the same integer. Suppose $P' \Downarrow (H, [], \rho', \texttt{return}\ x)$ and $\rho'(x) = i$. By induction on the number of rewriting steps using the Postponement Lemma, I can show that $P \Downarrow (H \uplus H_2, [], \rho', \texttt{return}\ x)$, and clearly $\rho'(x) = i$.

Now suppose $P \Downarrow (H, [], \rho', x)$ and $\rho'(x) = i$. By induction on the number of rewriting steps using the Diamond Lemma, we know that there exists a P'' such that $P' \Downarrow P''$ and and $P'' \overset{\text{trace}}{\longmapsto} (H, [], \rho', \texttt{return}\ x)$. Thus, $P'' = (H \uplus H', [], \rho', \texttt{return}\ x)$ and $\rho'(x) = i$ and both P and P' compute the same answer. $\square$

This theorem shows that a single application of tracing collection results in a Kleene-equivalent program. A real implementation interleaves garbage collection with evaluation. Let R stand for the standard set of rewriting rules (see Figure 7.3) and let T stand for this set with the tracing garbage collection rule. The following theorem shows that evaluation under R and T is equivalent.

Theorem 7.2.7 *For all P, $P \Downarrow_R (H, [], \rho, \texttt{return } x)$ iff $P \Downarrow_T (H', [], \rho, \texttt{return } x)$.*

Proof: Clearly any evaluation under the R rules can be simulated by the T rules simply by not performing any collection steps. Now suppose $P \Downarrow_T (H_1, [], \rho_1, \texttt{return } x_1)$ and $\rho_1(x_1) = i$. Then there exists a finite rewriting sequence using T as follows:

$$P \stackrel{\mathrm{T}}{\longmapsto} P_1 \stackrel{\mathrm{T}}{\longmapsto} P_2 \stackrel{\mathrm{T}}{\longmapsto} \cdots \stackrel{\mathrm{T}}{\longmapsto} (H_1, [], \rho_1, \texttt{return } x_1)$$

I can show by induction on the number of rewriting steps in this sequence, using the Postponement Lemma, that all garbage collection steps can be performed at the end of the evaluation sequence. This provides us with an alternative evaluation sequence where all the R steps are performed at the beginning:

$$\begin{array}{l} P \stackrel{\mathrm{R}}{\longmapsto} P'_1 \stackrel{\mathrm{R}}{\longmapsto} P'_2 \stackrel{\mathrm{R}}{\longmapsto} \cdots \stackrel{\mathrm{R}}{\longmapsto} P'_n \stackrel{\mathrm{trace}}{\longmapsto} \\ \quad P_{n+1} \stackrel{\mathrm{trace}}{\longmapsto} P_{n+2} \stackrel{\mathrm{trace}}{\longmapsto} \cdots \stackrel{\mathrm{trace}}{\longmapsto} (H_1, [], \rho_1, \texttt{return } x_1) \end{array}$$

Since collection does not affect the expression part of a program and only removes bindings from the heap, $P'_n = (H_1 \uplus H_2, [], \rho_1, \texttt{return } x_1)$ for some H_2. Therefore, $P \Downarrow_R P'_n$. Thus, any evaluation under T can be simulated by an evaluation under R. □

Finally, I can prove that tracing garbage collection is *optimal* with respect to my definition of garbage in the sense that it can collect as much garbage as any other collector can.

Theorem 7.2.8 (Tracing Collection Optimal) *If P' is a collection of a well-typed program P, then $P \stackrel{\mathrm{trace}}{\longmapsto} P'$.*

Proof: Let $P = (H \uplus H', S, \rho, e)$ and suppose $P' = (H, S, \rho, e)$ is a collection of P. By the definition of heap garbage, P' is well-typed. Hence, P' is closed and $FL_{\mathrm{prog}}(P') = \emptyset$. Thus $P \stackrel{\mathrm{trace}}{\longmapsto} P'$. □

7.3 Type-Directed Garbage Collection

In the previous section, I gave a specification for tracing garbage collection as a rewriting rule and showed that this new rewriting rule did not effect a program's evaluation. However, the rewriting rule is simply a specification and not an algorithm for computing a collection of a program. It assumes some mechanism for partitioning the set of bindings in the heap into two disjoint pieces, such that one set of bindings is unreachable from the second set of bindings and the rest of the program. Real garbage collection algorithms need a deterministic mechanism for generating this partitioning. In this section I formulate an abstract version of such a mechanism, the tracing garbage collection algorithm, by lifting the ideas of mark/sweep and copying collectors to the level of program syntax.

The basic idea behind an algorithm for tracing garbage collectors is to calculate the set of locations accessible from the current context (i.e., the environment and stack). These locations must be preserved to keep the program closed. Next, for each location that has been preserved, we examine the heap value to which this location is bound. Each location within this heap value must also be preserved. We iterate this process until all locations in the heap have been classified as accessible or inaccessible.

How do we calculate the set of locations within an environment or stack or heap value? One approach is to deconstruct the object in question based on its abstract syntax and simply find all of the "l" objects. However, this requires that the distinctions between syntactic classes remain apparent at runtime. That is, we must be able to tell l objects from f and i and $\langle\rangle$ objects and we must be able to break tuples and closures into their components, determine which components are l objects, etc. Fundamentally, this is a parsing problem: To deconstruct an object to find its location components, we must leave enough markers or *tags* in the representation of objects to determine what the components of the object are.

Tagging objects directly is unattractive because it can cost both space and time during computation. For example, if we tag integer and floating point values so that we can tell them from locations, then we can no longer directly use the machine's primitive operations, such as addition, to implement our primitive integer and floating point operations. Instead, we must strip the tag(s) from the value, apply the machine operation, and then tag the result.

An alternative approach to tagging values is to use types to guide the process of finding the locations in an object. In particular, by the Canonical Forms Lemma, we know that if we have an answer of the form $(H, S, \rho, \texttt{return}\ x)$ of type $\langle\tau_1 \times \tau_2\rangle$, then x is bound to some location l in ρ. Furthermore, we know that $H(l)$ is defined and is of the form $\langle v_1, v_2\rangle$. If τ_i is a tuple type, code type, or arrow type, then we know that v_i is a location. In this fashion, given the type of an object in the heap, we can extract all the locations in that object.

Extracting locations from closures requires a bit more cooperation from the implementation. In particular, we must assume that all closures provide sufficient information for finding their environment components and the type of the environment. However, once this information is in hand, we can use the type of the environment argument of the code to determine all of the locations that are in the closure's environment.

I formalize the process of extracting locations based on types as follows. First, I define a subset of types corresponding to heap values:

$$\text{(pointer types)} \quad \phi \ ::= \ \langle\tau_1 \times \tau_2\rangle \mid \mathsf{code}(\tau_{\text{env}}, \tau) \mid \tau_1 \to \tau_2$$

Next, I construct a *partial* function, TL_{hval}, that maps a location, a pointer type, and a heap to a location type assignment. I use TL to remind the reader that we are extracting a set of Typed Locations. Since code heap values never have free locations, the definition of TL_{hval} at code types is simply the empty type assignment:

$$TL_{\text{hval}}[l{:}\mathsf{code}(\tau_{\text{env}}, \tau), H] = \emptyset$$

The free locations of a tuple are those components whose types are pointer types.

$$\begin{array}{l}
TL_{\text{hval}}[l{:}\langle\phi_1 \times \phi_2\rangle, H \uplus \{l{=}\langle v_1, v_2\rangle\}] = \{v_1{:}\phi_1\} \cup \{v_2{:}\phi_2\} \\
TL_{\text{hval}}[l{:}\langle\phi_1 \times \tau_2\rangle, H \uplus \{l{=}\langle v_1, v_2\rangle\}] = \{v_1{:}\phi_1\} \\
TL_{\text{hval}}[l{:}\langle\tau_1 \times \phi_2\rangle, H \uplus \{l{=}\langle v_1, v_2\rangle\}] = \{v_2{:}\phi_2\} \\
TL_{\text{hval}}[l{:}\langle\tau_1 \times \tau_2\rangle, H \uplus \{l{=}\langle v_1, v_2\rangle\}] = \emptyset
\end{array}$$

The free locations of a closure include the location bound to the code and possibly the environment value. To determine if the environment is a location and to determine its type, we must look at the type ascribed to the environment argument of the code. If this type is a pointer type ϕ, then the environment component of the closure must be a location whose contents are described by ϕ.

$$TL_{\text{hval}}[l{:}\tau_1 \to \tau_2, H] = \{l_{\text{code}}{:}\mathsf{code}(\phi, \tau_1 \to \tau_2), v_{\text{env}}{:}\phi\}$$

$$\begin{array}{l}
\text{when } H(l) = \langle\langle l_{\text{code}}, v_{\text{env}}\rangle\rangle \\
\text{and } H(l_{\text{code}}) = \mathtt{vcode}[x_{\text{env}}{:}\phi, x{:}\tau_1].e.
\end{array}$$

Otherwise, if the type of the environment argument is not a pointer type, then the environment of the closure is not a location and thus only the code pointer is in the resulting type assignment:

$$TL_{\text{hval}}[l{:}\tau_1 \to \tau_2, H] = \{l_{\text{code}}{:}\mathsf{code}(\tau_{\text{env}}, \tau_1 \to \tau_2)\}$$

$$\begin{array}{l}
\text{when } H(l) = \langle\langle l_{\text{code}}, v_{\text{env}}\rangle\rangle \\
\text{and } H(l_{\text{code}}) = \mathtt{vcode}[x_{\text{env}}{:}\tau_{\text{env}}, x{:}\tau_1].e
\end{array}$$

The following lemma shows that $TL_{\text{hval}}[l{:}\Psi(l), H]$ is always defined and consistent with Ψ whenever Ψ describes the contents of the heap H. Thus, if we know the type of some location, then we can always extract the pointers contained in the heap value bound to that location.

Lemma 7.3.1 (Canonical Heap Values) *If* $\vdash_{\text{heap}} H : \Psi \uplus \{l{:}\tau\}$*, then* $TL_{\text{hval}}[l{:}\tau, H] = \Psi_h$ *for some* Ψ_h *and* $\Psi_h \subseteq \Psi \uplus \{l{:}\tau\}$.

Proof (sketch): By examination of the heap, heap value, and small value typing rules. □

The TL_{hval} function provides the functionality that we need to keep a garbage collector running. All that remains is to extract the locations and their types from the stack and environment of a program.

$$TL_{\text{env}}\{x_1{:}\tau_1{=}v_1, \cdots, x_n{:}\tau_n{=}v_n\}] \;=\; \cup_{i=1}^{n}\{v_i{:}\tau_i \mid \exists\phi.\tau_i = \phi\}$$

$$\begin{aligned} TL_{\text{stack}}([]) &= \emptyset \\ TL_{\text{stack}}(S[\rho, \lambda x{:}\tau.\, e]) &= TL_{\text{stack}}(S) \cup TL_{\text{env}}(\rho) \end{aligned}$$

With these functions in hand, I can now construct an algorithm that finds all of the locations that must be preserved in a program. The algorithm is formulated as a rewriting system between triples consisting of a heap, a location type assignment, and another heap, (H_f, Ψ, H_t). In traditional garbage collection terminology, the first heap is termed the "from-heap" or "from-space", the location type assignment is called the "scan-set" or "frontier", and the second heap is called the "to-heap" or "to-space".

Initially, the from-space contains all of the bindings from the program's heap; when the algorithm terminates, it contains those bindings that do not need to be preserved. Correspondingly, the to-space is initially empty; when the algorithm terminates, it contains all of the bindings that must be preserved. During each step of the algorithm, the scan-set contains the locations and their types that are bound in the from-space but are immediately reachable from the to-space. The scan-set is initialized by finding the locations in the current environment and stack.

The body of the algorithm proceeds as follows: a location l of type ϕ is removed from Ψ. If l is bound in the from-space to a heap value, then we use $TL_{\text{hval}}[l{:}\phi, H]$ to extract the locations contained in $H(l)$, where H is the union of the from- and to-spaces. For each such location l', we check to see if l' has already been forwarded to the to-set H_t. Only if l' is not bound in H_t do we add the location and its type to the scan-set Ψ. This ensures that a variable moves at most once from the from-space to the scan-set. I formalize this process via the following rewriting rule:

$$(H_f \uplus \{l{=}h\}, \Psi_s \uplus \{l{:}\phi\}, H_t) \Rightarrow (H_f, \Psi_s \cup \Psi'_s, H_t \uplus \{l{=}h\})$$

$$\text{where } \Psi'_s = \{l'{:}\phi' \in TL_{\text{hval}}[l{:}\phi, H_f \uplus \{l{=}h\} \uplus H_t] \mid l' \notin \mathit{Dom}(H_t) \uplus \{l\}\}$$

Once the scan-set becomes empty, the algorithm terminates and the to-space is taken as the new, garbage-collected heap of the program, while the from-space is discarded. The initialization and finalization steps are captured by the following inference rule:

$$\frac{(H, TL_{\text{env}}(\rho) \cup TL_{\text{stack}}(S), \emptyset) \Rightarrow^{*} (H_f, \emptyset, H_t)}{(H, S, \rho, e) \overset{\text{tr-alg}}{\longmapsto} (H_t, S, \rho, e)}$$

To prove the correctness of this garbage collection algorithm, it suffices to show that, whenever $P \overset{\text{tr-alg}}{\longmapsto} P'$, then $P \overset{\text{trace}}{\longmapsto} P'$, since I have already shown that the tracing garbage collection specification is correct. However, to ensure that we have a proper algorithm, I must also show that there always exists some P' such that $P \overset{\text{tr-alg}}{\longmapsto} P'$. That is, I must show that the algorithm does not get stuck.

I begin by establishing a set of invariants, with respect to the original program, that are to be maintained by the algorithm. I note that if a program $P = (H, S, \rho, e)$ is well-typed, then there is a unique location type assignment Ψ_P, variable type assignment Γ_P, and unique types τ_P and τ'_P such that: $\emptyset \vdash_{\text{heap}} H : \Psi_P$, $\Psi_P \vdash_{\text{stack}} \tau'_P \to \tau_P$, $\Psi_P \vdash_{\text{env}} \rho : \Gamma_P$ and $\Gamma_P \vdash_{\text{exp}} e : \tau'_P$. Hence, for a well-typed P, I write Ψ_P, Γ_P, τ_P, and τ'_P to represent these respective objects.

Definition 7.3.2 (Well-Formedness) *Let $P = (H, S, \rho, e)$ be a well-typed program. The tuple (H_f, Ψ_s, H_t) is well-formed with respect to P iff, taking $\Psi_t = \{l{:}\Psi_P(l) \mid l \in Dom(H_t)\}$ and $\Psi_f = \{l{:}\Psi_P(l) \mid l \in Dom(H_f)\}$:*

1. $H_f \uplus H_t = H$
2. $\Psi_s \subseteq \Psi_f$
3. $\Psi_s \uplus \Psi_t \vdash_{\text{stack}} S : \tau'_P \to \tau_P$
4. $\Psi_s \uplus \Psi_t \vdash_{\text{env}} \rho : \Gamma_P$
5. $\Psi_s \vdash_{\text{heap}} H_t : \Psi_t$.

Roughly speaking, the invariants ensure that: (1) all of the heap bindings are accounted for in either the from-space or the to-space and these two spaces are disjoint, (2) the scan-set types some of the locations in the from-space and these types are consistent with the rest of the program, (3) the scan-set coupled with the types of locations in the to-space allow us to type the stack appropriately, (4) the scan-set coupled with the types of locations in the to-space allow us to type the environment appropriately, and (5) the scan-set allows us to type the to-space appropriately. The following lemma shows that these invariants are preserved by the algorithm.

Lemma 7.3.3 (Preservation) *If $\vdash_{\text{prog}} P : \tau_P$, (H_f, Ψ_s, H_t) is well-formed with respect to P and $(H_f, \Psi_s, H_t) \Rightarrow (H'_f, \Psi'_s, H'_t)$, then (H'_f, Ψ'_s, H'_t) is well-formed with respect to P.*

Proof: Suppose $(H_f \uplus \{l{=}h\}, \Psi_s \uplus \{l{:}\tau\}, H_t)$ is well-formed with respect to $P = (H, S, \rho, e)$ and suppose $(H_f \uplus \{l{=}h\}, \Psi_s \uplus \{l{:}\tau\}, H_t) \Rightarrow (H_f, \Psi_s \uplus \Psi'_s, H_t \uplus \{l{=}h\})$ where $\Psi'_s = \{l'{:}\tau' \in TL_{\text{hval}}[l{:}\tau, H_f \uplus \{l{=}h\} \uplus H_t] \mid l' \notin Dom(H_t)\}$.

By condition (1), $H = (H_f \uplus \{l{=}h\}) \uplus H_t$ thus, $H = H_f \uplus (H_t \uplus \{l{=}h\})$.

By condition (2), we know that $l{:}\tau \in \Psi_P$ and $\Psi_s \subseteq \Psi_P$. By the Canonical Heap Values Lemma, $TL_{\text{hval}}[\tau](h) \subseteq \Psi_P$. Hence, $\Psi'_s \subseteq \Psi_s \uplus \Psi'_s \subseteq \Psi_P$.

By conditions (3) and (4), taking $\Psi = (\Psi_s \uplus \{l{:}\tau\}) \uplus \Psi_t$, we know that $\Psi \vdash_{\text{stack}} S : \tau'_P \to \tau_P$ and $\Psi \vdash_{\text{env}} \rho : \Gamma_P$. By condition (2) we know that $l{:}\tau \in \Psi_P$. Thus, $\Psi_t \uplus \{l{:}\tau\}$ is well-formed and taking $\Psi' = (\Psi_s \cup \Psi'_s) \uplus (\Psi_t \uplus \{l{:}\tau\})$, we have $\Psi' \vdash_{\text{stack}} S : \tau'_P \to \tau_P$ and $\Psi' \vdash_{\text{env}} \rho : \Gamma_P$.

By condition (5), $\Psi_s \uplus \{l{:}\tau\} \vdash_{\text{heap}} H_t : \Psi_t$. I must show $(\Psi_s \cup \Psi'_s) \vdash_{\text{heap}} H_t \uplus \{l{=}h\} : \Psi'_t$ where $\Psi_t \uplus \{l{:}\tau\}$. By the Canonical Heap Values Lemma, $TL_{\text{hval}}[l{:}\tau, H] \vdash_{\text{hval}} h : \tau$, where $H = H_f \uplus \{l{=}h\} \uplus H_t$. Hence, $(\Psi_s \cup \Psi'_s) \uplus \Psi_t \vdash_{\text{hval}} h : \tau$. Thus, by the heap typing rule, $(\Psi_s \cup \Psi'_s) \vdash_{\text{heap}} H_t \uplus \{l{=}h\} : \Psi_t \uplus \{l{:}\tau\}$. □

The following lemma shows that at each step in the algorithm, either the scan-set is empty — in which case the algorithm is finished — or else the algorithm can step to a new state.

Lemma 7.3.4 (Progress) *If $\vdash_{\text{prog}} P : \tau_P$ and (H_f, Ψ_s, H_t) is well-formed with respect to P, then either Ψ_s is empty or else $(H_f, \Psi_s, H_t) \Rightarrow (H'_f, \Psi'_s, H'_t)$ for some (H'_f, Ψ'_s, H'_t).*

Proof: Suppose Ψ_s contains the binding $l{:}\tau$ and let $H = H_f \uplus H_t$. First, I must show that l is bound to some heap value in H_f and that heap value has a shape described by τ. By the second requirement of well-formedness and the definition of Ψ_f, we know that $\Psi_f(l) = \Psi_P(l) = \tau$. Since $\vdash_{\text{heap}} H : \Psi_P$, by the Canonical Heap Values Lemma, we know that $H(l) = h$ for some h, $TL_{\text{hval}}[l{:}\tau, H] = \Psi_h$, and $\Psi_h \subseteq \Psi_P$. By the definition of Ψ_f and the first requirement of well-formedness, the binding $l{=}h$ must be in H_f.

Taking $\Psi''_s = \{l'{:}\tau' \in \Psi_h \mid l' \notin Dom(H_t)\}$, I must now show that $(\Psi_s \setminus \{l{:}\tau\}) \cup \Psi''_s$ is a valid location type assignment. But since $\Psi_h \subseteq \Psi_P$, then $\Psi''_s \subseteq \Psi_P$. By the second requirement of well-formedness, $\Psi_s \subseteq \Psi_f \subseteq \Psi_P$, so $(\Psi_s \setminus \{l{:}\tau\}) \cup \Psi''_s$ is well formed as a subset of Ψ_P.

Finally, since H_f and H_t are disjoint but cover H, l cannot be bound in H_t and hence $H_t \uplus \{l{=}h\}$ is well-formed. Thus, taking $H'_f = H_f \setminus \{l{=}h\}$, $\Psi'_s = \Psi_s \setminus \{l{:}\tau\} \cup \Psi''_s$, and $H'_t = H_t \uplus \{l{=}h\}$, we know that $(H_f, \Psi_s, H_t) \Rightarrow (H'_f, \Psi'_s, H'_t)$. □

With the Preservation and Progress Lemmas in hand, I can establish the correctness of the algorithm.

Theorem 7.3.5 (Tracing Algorithm Correctness) *If P is well-typed, then there exists a P' such that $P \stackrel{\text{tr-alg}}{\longmapsto} P'$ and $P \simeq P'$.*

Proof: Let $P = (H, S, \rho, e)$. First, I must show that $(H, TL_{\text{env}}(\rho) \cup TL_{\text{stack}}(S), \emptyset)$ exists and is well-formed with respect to P. Since the to-space is empty, conditions one and five of well-formedness are trivially satisfied. By the env typing rule, it is clear that $TL_{\text{env}}(\rho)$ exists and is a subset of Ψ_P. By the stack typing rules and the env typing rule, it is clear that $TL_{\text{stack}}(S)$ exists and is a subset of Ψ_P. Hence, $TL_{\text{env}}(\rho) \cup TL_{\text{stack}}(S)$ is a well-formed location type assignment that is a subset of Ψ_P. Furthermore, it is clear that all of the free locations in both the stack and environment are contained in this location type assignment. Hence, conditions two, three, and four are satisfied and the initial tuple is well-formed with respect to P.

Since $(H, TL_{\text{env}}(\rho) \cup TL_{\text{stack}}(S), \emptyset)$ is well-formed, by progress, the algorithm will continue to run until the scan-set is empty, at which point the algorithm terminates in the state $(H_f, \emptyset, H_t)$. By preservation, we know that this tuple is well-formed with respect to P. Hence, we know that, taking $\Psi_t = \{l{:}\tau \mid l \in \Psi_P\}$, $\Psi_t \vdash_{\text{stack}} S : [\tau'_P] \to \tau_P$, $\Psi_t \vdash_{\text{env}} \rho : \Gamma_P$, and $\vdash_{\text{heap}} H_t : \Psi_t$. Consequently, the program $P' = (H_t, S, \rho, e)$ is closed and thus $P \overset{\text{trace}}{\longmapsto} P'$. By the correctness of the tracing specification, we know that $P \simeq P'$. Thus, P' is a collection of P. □

Finally, the tracing algorithm that I have presented is optimal with respect to my definition of garbage.

Theorem 7.3.6 *Let* $P \overset{\text{tr-alg}}{\longmapsto} (H_1, S, \rho, e)$*. If* (H_2, S, ρ, e) *is a collection of* P *then* $H_1 \subseteq H_2$*.*

Proof: By theorem 7.2.8, it suffices to show that if $P \overset{\text{trace}}{\longmapsto} P'$ where $P' = (H_2, S, \rho, e)$, then $H_1 \subseteq H_2$. Let $l{=}h$ be a binding in H_1. I must show that this binding is also in H_2. I do so by analyzing how l is placed in the scan set and hence forwarded from the from-space to the to-space during the execution of the tracing algorithm.

If l is placed in the initial scan-set, then l occurs free in either the range of ρ or else the range of the environment of some closure on the stack. Hence, l must be bound in H_2 to keep P' closed.

Suppose l is is placed in the scan-set because it is found via $TL_{\text{hval}}[\tau'](h)$ for some $l'{:}\tau'$ already in the scan-set. The binding, $l'{=}h'$ is forwarded to the to-space. Therefore, h' is in the range of H_2. But then $FL_{\text{hval}}(h')$ contains l. Thus, for P' to be closed, H_2 must contain the binding for l. □

7.4 Generational Collection

The tracing garbage collection algorithm I presented in the previous section examines *all* of the reachable bindings in the heap to determine that the rest of the bindings may be removed. By carefully partitioning the heap into smaller heaps, a garbage collector can scan less than the whole heap and still free significant amounts of memory. A *generational partition* of a program's heap is a sequence of sub-heaps ordered in such a way that "older" generations never have pointers to "younger" generations.

Definition 7.4.1 (Generational Partition) *A* generational partition *of a heap H is a sequence of heaps $H_1, H_2, \ldots, H_n$ such that $H = H_1 \uplus H_2 \uplus \cdots \uplus H_n$ and for all i such that $1 \leq i < n$, $FL_{\mathrm{heap}}(H_i) \cap Dom(H_{i+1} \uplus H_{i+2} \uplus \cdots \uplus H_n) = \emptyset$. The H_i are referred to as generations and H_i is said to be an older generation than H_j if $i < j$.*

Given a generational partition of a program's heap, a tracing garbage collector can eliminate a set of bindings in younger generations without looking at any older generations.

Theorem 7.4.2 (Generational Collection) *Let $H_1, \ldots, H_i, \ldots, H_n$ be a generational partition of the heap of $P = (H, S, \rho, e)$. Suppose $H_i = (H_i^1 \uplus H_i^2)$ and $Dom(H_i^2) \cap FL_{\mathrm{prog}}(H_i^1 \uplus H_{i+1} \uplus \cdots \uplus H_n, S, \rho{:}\Gamma, e) = \emptyset$. Then $P \stackrel{\mathrm{trace}}{\longmapsto} (H \setminus H_i^2, S, \rho, e)$.*

Proof: I must show that $Dom(H_i^2) \cap FL_{\mathrm{prog}}(H \setminus H_i^2, S, \rho, e) = \emptyset$. Since $H_1, \cdots, H_n$ is a generational partition of H, for all j, $1 \leq j < i$, $FL_{\mathrm{heap}}(H_j) \cap Dom(H_{j+1} \uplus \cdots \uplus H_n) = \emptyset$. Hence, $FL_{\mathrm{heap}}(H_1 \uplus \cdots \uplus H_{i-1}) \cap Dom(H_i^2) = \emptyset$. Now,

$$
\begin{array}{rl}
& FL_{\mathrm{prog}}(H \setminus H_i^2, S, \rho, e) \cap Dom(H_i^2) \\
= & (FL_{\mathrm{heap}}(H \setminus H_i^2) \cup FL_{\mathrm{stack}}(S) \cup FL_{\mathrm{env}}(\rho) \cup FL_{\mathrm{exp}}(e)) \cap Dom(H_i^2) \\
= & (FL_{\mathrm{heap}}(H_1 \uplus \cdots \uplus H_{i-1}) \cup FL_{\mathrm{heap}}(H_i^1 \uplus \cdots \uplus H_n) \cup \\
& \qquad FL_{\mathrm{stack}}(S) \cup FL_{\mathrm{env}}(\rho) \cup FL_{\mathrm{exp}}(e)) \cap Dom(H_i^2) \\
= & (FL_{\mathrm{heap}}(H_1 \uplus \cdots \uplus H_{i-1}) \cap Dom(H_i^2)) \cup \\
& \quad ((FL_{\mathrm{heap}}(H_i^1 \uplus \cdots \uplus H_n) \cup FL_{\mathrm{stack}}(S) \cup FL_{\mathrm{env}}(\rho) \cup FL_{\mathrm{exp}}(e)) \cap Dom(H_i^2)) \\
= & \emptyset \cup ((FL_{\mathrm{heap}}(H_i^1 \uplus \cdots \uplus H_n) \cup FL_{\mathrm{stack}}(S) \cup FL_{\mathrm{env}}(\rho) \cup FL_{\mathrm{exp}}(e)) \cap Dom(H_i^2)) \\
= & FL_{\mathrm{prog}}(H_i^1 \uplus \cdots \uplus H_n, S, \rho, e) \cap Dom(H_i^2) \\
= & \emptyset
\end{array}
$$

□

Generational collection is important for three practical reasons: first, evaluation of programs makes it easy to maintain generational partitions.

Theorem 7.4.3 (Generational Preservation) *Let $P = (H, S, \rho, e)$ be a well-typed program. If $H_1, \ldots, H_n$ is a generational partition of H and $P \longmapsto (H \uplus H', S, \rho, e)$, then $H_1, \ldots, H_n, H'$ is a generational partition of $H \uplus H'$.*

Proof: The only evaluation rules that modify the heap are the rules that allocate tuples and closures. The other rules leave the heap intact and hence preserve the partition trivially. Since the allocation rules only add a binding to the heap and do not modify the rest of the heap, all I must show is that there are no references in the older generations to this new location. But this must be true since a new location is chosen for the allocated heap value. □

Clearly, the addition of certain language features such as assignment or memoization breaks the Generational Preservation Theorem. The problem with these features is that bindings in the heap can be updated so that a heap value in an older generation contains a reference to a location in a younger generation. It is possible to maintain a generational partition for such languages by keeping track of all older bindings that are updated and by moving them from the older generation to a younger generation. The mechanism that tracks updates to older generations is called a *write barrier*. Wilson's overview provides many examples of techniques used to implement write barriers [128].

The second reason generational collection is important is that, given a generational partition, we can directly use the tracing collection algorithm to generate a *generational* collection of a program.

The following generational collection rule starts simply forwards the entire older generation at the beginning of the algorithm and then processes the younger generation.

$$\frac{H_1; H_2 \text{ a generational partition} \qquad (H_2, \{l{:}\tau \in TL_{\text{stack}}(S) \cup TL_{\text{env}}(\rho) \mid l \notin Dom(H_1)\}, H_1) \Rightarrow (H_f, \emptyset, H_1 \uplus H_2')}{(H_1 \uplus H_2, S, \rho, e) \overset{\text{gen-alg}}{\longmapsto} (H_1 \uplus H_2', S, \rho, e)}$$

The rule's soundness follows directly from the Generational Collection Theorem, as well as the soundness of the tracing collection algorithm.

The third reason generational collection is important is that empirical evidence shows that "objects tend to die young" [120]. That is, recently allocated bindings are more likely to become garbage in a small number of evaluation steps. Thus, if we place recently allocated bindings in younger generations, we can concentrate our collection efforts on these generations, ignoring older generations, and still eliminate most of the garbage.

7.5 Polymorphic Tag-Free Garbage Collection

In this section, I show how to apply type-based, tag-free garbage collection to a λ_i^{ML}-based language called λ_i^{ML}-GC. It is possible to give a low-level operational semantics for λ_i^{ML} in the style of Mono-GC, where environments and the stack are made explicit. However, the type structure of λ_i^{ML} is considerably more complex than for Mono-GC, and as a result, proving even relatively basic properties, such as type preservation, is considerably more difficult than in the simply-typed setting. Consequently, I use a somewhat higher-level semantics to describe evaluation of λ_i^{ML}-GC programs. This semantics is a cross between the contextual rewriting semantics used in earlier chapters and the allocation

(kinds)	κ	$::=$	$\Omega \mid \kappa_1 \to \kappa_2$
(con)	μ	$::=$	$(\nu :: \kappa)$
(raw con)	ν	$::=$	$t \mid u \mid \mathsf{Arrow}(\mu_1, \mu_2) \mid \lambda t{::}\kappa.\mu \mid \mu_1\, \mu_2 \mid \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i; \mu_a)$
(small con)	u	$::=$	$l \mid \mathsf{Int}$
(heap con)	q	$::=$	$\mathsf{Arrow}(u_1, u_2) \mid \lambda t{::}\kappa.\mu$
(con heap)	Q	$::=$	$\{l_1{=}q_1, \cdots, l_n{=}q_n\}$
(types)	σ	$::=$	$T(\mu) \mid \mathsf{int} \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma$
(norm types)	ς	$::=$	$\mathsf{int} \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma$
(exp)	e	$::=$	$x \mid (v : \varsigma) \mid (r : \sigma)$
(raw exp)	r	$::=$	$\lambda x{:}\sigma.e \mid e_1\, e_2 \mid \Lambda t{::}\kappa.e \mid e\,[\mu] \mid \mathtt{typerec}\ \mu\ \mathtt{of}\ [t.\sigma](e_i; e_a)$
(small val)	v	$::=$	$i \mid l$
(heap val)	h	$::=$	$\lambda x{:}\sigma.e \mid \Lambda t{::}\kappa.e$
(val heap)	H	$::=$	$\{l_1{=}h_1, \cdots, l_n{=}h_n\}$
(program)	P	$::=$	(Q, H, e)
(answer)	P	$::=$	$(Q, H, v{:}\varsigma)$

Figure 7.7: Syntax of λ_i^{ML}-GC

semantics of Mono-GC. Heaps are left explicit, but the stack and environments are implicitly represented by evaluation contexts and meta-level substitution of small values for variables. The resulting system abstracts enough details that proofs are tractable, yet exposes the key issues of tag-free collection.

7.5.1 λ_i^{ML}-GC

The syntax of λ_i^{ML}-GC is given in Figure 7.7. Programs consist of two heaps and an expression. The Q heap maps locations to constructor heap values, whereas the H heap maps locations to expression heap values. In practice, one heap suffices for both constructors and expressions, but making a distinction simplifies the static semantics for the language. I assume that the constructor heap of a program contains no cycles (this is reflected in the static semantics below), but make no such assumption regarding the expression heap. The expression of the program can refer directly to locations bound in the heaps, instead of indirecting through an environment. As with Mono-GC, I consider programs to be equivalent up to reordering and α-conversion of locations bound in the heap and variables bound in constructors and expressions.

Each raw constructor ν is labelled with its kind, and almost all raw expressions are labelled with types. This information corresponds to the kind or type information that would be present on bound variables of a `let`-expression in an A-normal form. By labelling nested computations with kind or type information at compile time, we effectively assign kinds/types to any intermediate values allocated during computation. Constructor values and expression values in the heaps are *not* paired with summary kind or type information; this information is recovered during garbage collection from the information labelling computations.

During garbage collection, we need to determine shape information concerning values found in computations from the kinds and types labelling these values. Unfortunately, it is not always possible to determine an expression value's shape from its type. In particular, if the type is of the form $T(\mu)$ where μ is some constructor computation, then we must "run" the computation to reach at least a head-normal form, either Int or $\mathsf{Arrow}(\mu_1, \mu_2)$, in order to determine the shape of the object[2]. But running this constructor computation during garbage collection is problematic because the computation may need to allocate values. It seems as though to free storage, we must garbage collect, but to garbage collect, we might need to allocate more storage.

The solution to this problem is to "run" the necessary constructors within types during evaluation and ensure that all needed types are always in head-normal form when garbage collection is invoked. This constraint is reflected in the syntax of λ_i^{ML}-GC by the fact that only ς types (types in head-normal form) can label small values. The evaluation rules for λ_i^{ML}-GC (see below) ensure that this constraint is maintained at all times. In particular, small values and their normalized type labels are substituted for free variables during evaluation. In the TIL compiler (see Chapter 8), this constraint is maintained by explicitly reifying type computations and by labelling variables of unknown shape with a reified type. If λ_i^{ML} did not have a phase distinction between types and terms (i.e., if types were dependent upon terms in some fashion), then we could not guarantee that all needed types would be computed before garbage collection was invoked.

Figure 7.8 gives the evaluation contexts and instructions for the constructors, types, and terms of λ_i^{ML}-GC, and Figures 7.9, and 7.10 give the rewriting rules for the language. As in Mono-GC, large values are allocated on the heap and replaced with a reference to the appropriate location. Unlike Mono-GC, I use evaluation contexts to determine the next instruction instead of relying upon an A-normal form to provide explicit sequencing and a stack to represent the continuation. Furthermore, I use meta-level substitution of small values for variables at function application, instead of installing these values in

[2]Head-normal forms are not always sufficient to determine shape. For example, components of pair types must also be normalized so that we can determine which components of the pair are locations.

$$\begin{array}{llll}
\text{(con ctxt)} & U & ::= & [] \mid (N :: \kappa) \\
\text{(raw ctxt)} & N & ::= & \mathsf{Arrow}(U, \mu) \mid \mathsf{Arrow}(u{::}\kappa, U) \mid U\, \mu \mid (u{::}\kappa)\, U \mid \\
 & & & \mathsf{Typerec}\ U\ \mathsf{of}\ (\mu_i; \mu_a) \\
\text{(con instr)} & J & ::= & \mathsf{Arrow}(u_1{::}\kappa_1, u_2{::}\kappa_2) \mid \lambda t{::}\kappa.\mu \mid (l{::}\kappa_1)\,(u_2{::}\kappa_2) \mid \\
 & & & \mathsf{Typerec}\ (u{::}\kappa)\ \mathsf{of}\ (\mu_i; \mu_a) \\
 & & & \\
\text{(type instr)} & K & ::= & T(u{::}\kappa) \mid T(U[J{::}\kappa]) \\
 & & & \\
\text{(exp ctxt)} & E & ::= & [] \mid (R : \sigma) \\
\text{(raw ctxt)} & R & ::= & E\, e \mid (l{:}\varsigma)\, E \mid E\, [\mu] \\
\text{(exp instr)} & I & ::= & (\lambda x{:}\sigma.e) : K \mid (\lambda x{:}\sigma.e) : \varsigma \mid (\Lambda t{::}\kappa.e) : \varsigma \mid ((l{:}\varsigma)\,(v{:}\varsigma')) : \sigma \mid \\
 & & & ((l{:}\varsigma)\, U[J{::}\kappa]) : \sigma \mid ((l{:}\varsigma)\, [u{::}\kappa]) : \sigma \mid \\
 & & & (\mathtt{typerec}\ U[J{::}\kappa]\ \mathtt{of}\ [t.\sigma'](e_i; e_a)) : \sigma \mid \\
 & & & (\mathtt{typerec}\ (u{::}\kappa)\ \mathtt{of}\ [t.\sigma'](e_i; e_a)) : \sigma \mid
\end{array}$$

Figure 7.8: λ_i^{ML}-GC Evaluation Contexts and Instructions

an environment. This avoids the need to assume closure conversion, greatly simplifying both the dynamic and static semantics for the language.

Note that evaluation of λ-expressions at the term level proceeds in two stages: first, the type labelling the expression is evaluated. Second, the λ-expression is bound to a new location on the heap and is replaced with this location within the expression. There is no need to evaluate the type labelling a Λ-expression, since this type must always begin with $\forall$ and hence is already in head-normal form. The rest of the rewriting rules are fairly standard and reflect the left-to-right, call-by-value evaluation strategy of the language.

It is fairly easy to see that evaluation of λ_i^{ML}-GC programs preserves the cycle-freedom of the constructor heap and that, given a cycle-free expression heap, evaluation preserves cycle-freedom.

7.5.2 Static Semantics of λ_i^{ML}-GC

In the description of the static semantics for λ_i^{ML}-GC, I use the following meta-variables to range over various sorts of assignments, mapping variables or locations to kinds and

1. $(Q, \mathsf{Arrow}(u_1{::}\kappa_1, u_2{::}\kappa_2) :: \kappa) \longmapsto (Q \uplus \{l{=}\mathsf{Arrow}(u_1, u_2)\}, l{::}\kappa)$

2. $(Q, (\lambda t{::}\kappa_1.\mu) :: \kappa) \longmapsto (Q \uplus \{l{=}\lambda t{::}\kappa_1.\mu\}, l{::}\kappa)$

3. $(Q, (l{::}\kappa_1)\,(u{::}\kappa_2)) \longmapsto (Q, \{u/t\}\mu) \qquad (Q(l) = \lambda t{::}\kappa.\mu)$

4. $(Q, (\mathsf{Typerec}\ (\mathsf{Int}{::}\kappa')\ \mathsf{of}\ (\mu_i; \mu_a)) :: \kappa) \longmapsto (Q, \mu_i)$

5. $$\begin{array}{l} (Q, (\mathsf{Typerec}\ (l{::}\kappa')\ \mathsf{of}\ (\mu_i; \mu_a)) :: \kappa) \longmapsto \\ \quad (Q, ((((\mu_a\,(u_1{::}\Omega) :: \Omega \to \kappa \to \kappa \to \kappa) \\ \qquad (u_2{::}\Omega) :: \kappa \to \kappa \to \kappa) \\ \qquad (\mathsf{Typerec}\ (u_1{::}\Omega)\ \mathsf{of}\ (\mu_i; \mu_a) :: \kappa) :: \kappa \to \kappa) \\ \qquad (\mathsf{Typerec}\ (u_2{::}\Omega)\ \mathsf{of}\ (\mu_i; \mu_a) :: \kappa)) :: \kappa) \qquad (Q(l) = \mathsf{Arrow}(u_1, u_2)) \end{array}$$

6. $$\frac{(Q, J{::}\kappa) \longmapsto (Q', \mu)}{(Q, U[J{::}\kappa]) \longmapsto (Q', U[\mu])}$$

7. $(Q, T(\mathsf{Int}{::}\kappa)) \longmapsto (Q, \mathsf{int})$

8. $(Q, T(l{::}\Omega)) \longmapsto (Q, T(u_1{::}\kappa_1) \to T(u_2{::}\kappa_2)) \qquad (Q(l) = \mathsf{Arrow}(u_1, u_2))$

9. $$\frac{(Q, J{::}\kappa) \longmapsto (Q', \mu)}{(Q, S[T(U[J{::}\kappa])]) \longmapsto (Q', S[T(U[\mu])])}$$

Figure 7.9: λ_i^{ML}-GC Constructor and Type Rewriting Rules

10. $$\frac{(Q, K) \longmapsto (Q', \sigma)}{(Q, H, (\lambda x{:}\sigma'.e) : K) \longmapsto (Q, H, (\lambda x{:}\sigma'.e) : \sigma)}$$

11. $(Q, H, (\lambda x{:}\sigma.e) : \varsigma) \longmapsto (Q, H \uplus \{l{=}\lambda x{:}\sigma.e\}, l{:}\varsigma)$

12. $(Q, H, (\Lambda t{::}\kappa.e) : \varsigma) \longmapsto (Q, H \uplus \{l{=}\Lambda t{::}\kappa.e\}, l{:}\varsigma)$

13. $(Q, H, ((l{:}\varsigma_1)\,(v{:}\varsigma)) : \sigma) \longmapsto (Q, H, \{(v{:}\varsigma)/x\}e)$ $\quad (H(l) = \lambda x{:}\sigma'.e)$

14. $$\frac{(Q, U[J{::}\kappa]) \longmapsto (Q', U[\mu])}{(Q, H, ((l{:}\varsigma)\,U[J{::}\kappa]) : \sigma) \longmapsto (Q', H, ((l{:}\varsigma)\,U[\mu]) : \sigma)}$$

15. $(Q, H, ((l{:}\varsigma_1)\,(u{::}\kappa)) : \sigma) \longmapsto (Q, H, \{u/t\}e)$ $\quad (H(l) = \Lambda t{::}\kappa.e)$

16. $$\frac{(Q, U[J{::}\kappa]) \longmapsto (Q', U[\mu])}{\begin{array}{c}(Q, H, (\texttt{typerec}\ U[J{::}\kappa]\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \sigma) \longmapsto \\ (Q', H, (\texttt{typerec}\ U[\mu]\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \sigma)\end{array}}$$

17. $(Q, H, (\texttt{typerec}\ (\mathsf{Int}{::}\kappa]\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \sigma) \longmapsto (Q, H, e_i)$

18. $$\begin{array}{l}(Q, H, (\texttt{typerec}\ (l{::}\kappa)\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \sigma) \longmapsto \\ \quad (Q, H, ((((((e_a\,(u_1{::}\Omega)) : \forall t_2{::}\Omega.\{u_1/t\}\sigma' \to \{t_2/t\}\sigma' \to \sigma) \\ \qquad (u_2{::}\Omega)) : \{u_1/t\}\sigma' \to \{u_2/t\}\sigma' \to \sigma \\ \qquad ((\texttt{typerec}\ (u_1{::}\Omega)\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \{u_1/t\}\sigma') : \{u_2/t\}\sigma' \to \sigma)) \\ \qquad ((\texttt{typerec}\ (u_2{::}\Omega)\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \{u_2/t\}\sigma')) : \sigma) \quad (Q(l) = \mathsf{Arrow}(u_1, u_2))\end{array}$$

19. $$\frac{(Q, H, I : \sigma) \longmapsto (Q', H', e)}{(Q, H, E[I : \sigma]) \longmapsto (Q', H', E[e])}$$

Figure 7.10: λ_i^{ML}-GC Expression Rewriting Rules

types:

$$\begin{array}{lrcl} \text{(variable kind assignment)} & \Delta & ::= & \{t_1{::}\kappa_1, \cdots, t_n{::}\kappa_n\} \\ \text{(variable type assignment)} & \Gamma & ::= & \{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\} \\ \text{(location kind assignment)} & \Pi & ::= & \{l_1{::}\kappa_1, \cdots, l_n{::}\kappa_n\} \\ \text{(location type assignment)} & \Psi & ::= & \{l_1{:}\varsigma_1, \cdots, l_n{:}\varsigma_n\} \end{array}$$

In addition I use Φ to range over maps from locations to both kinds and heap constructor values:

$$\Phi \quad ::= \quad \{l_1{::}\kappa_1{=}q_1, \cdots, l_n{::}\kappa_n{=}q_n\}$$

Given an assignment Φ, I use Q_Φ and Π_Φ to represent the heap and location kind assignment implicit in Φ.

The formation judgments for λ_i^{ML}-GC are as follows:

1.	$\Pi; \Delta \vdash \nu :: \kappa$	constructor formation
2.	$\Pi \vdash q :: \kappa$	constructor heap value formation
3.	$\Pi \vdash Q :: \Pi'$	constructor heap formation
4.	$\Pi \vdash \Phi$	kind and constructor assignment formation
5.	$\Pi; \Delta \vdash \sigma$	type formation
6.	$\Pi \vdash \Psi$	location type assignment formation
7.	$\Pi; \Delta \vdash \Gamma$	variable type assignment formation
8.	$\Phi; \Psi; \Delta; \Gamma \vdash e : \sigma$	expression formation
9.	$\Phi; \Psi \vdash h : \sigma$	heap value formation
10.	$\Phi; \Psi' \vdash H : \Psi$	heap formation
11.	$\vdash (Q, H, e) : \sigma$	program formation

The axioms and inference rules that allow us to derive these judgments are largely standard, so I will only provide a high-level overview. For judgments 1–7, Π tracks the kind of all free locations in the given constructor, constructor heap value, constructor heap, type, or assignment. For constructors and types, Δ tracks the kind of free type variables. The lack of Δ in the judgments 3–4 indicates that there can be no free type variables, only free locations within constructor heaps. Constructor heap formation consists of an axiom and an inference rule that allow us to construct heaps inductively, thereby ensuring the heap has no cycles:

$$\Pi \vdash \emptyset : \emptyset \qquad\qquad \frac{\Pi' \vdash Q :: \Pi \quad \Pi' \uplus \Pi \vdash q :: \kappa}{\Pi' \vdash Q \uplus \{l{=}q\} :: \Pi \uplus \{l{::}\kappa\}}$$

A kind and constructor assignment Φ is well-formed with respect to Π if $\Pi \vdash Q_\Phi :: \Pi_\Phi$.

Judgments 8–9 require more information than simply the kinds of free locations. This is because we must treat constructor locations as *transparent* type variables in order to recover the same definitional equivalence that we have in a totally substitution-based semantics. For example, if l_1 is bound to $\mathsf{Arrow}(u_1, u_2)$, then we must consider l_1 to be equivalent to $\mathsf{Arrow}(u_1, u_2)$. Therefore, equivalence of both constructors and types is given with respect to a kind and constructor assignment Φ:

12.	$\Phi; \Delta \vdash \mu \equiv \mu' :: \kappa$	constructor equivalence
13.	$\Phi; \Delta \vdash \sigma \equiv \sigma'$	type equivalence

The axioms and inference rules that allow us to conclude two constructors or types are equivalent are standard (see Chapter 3) with the addition of two axioms governing the transparency of locations:

$$\Phi \uplus \{l{::}\Omega{=}\mathsf{Arrow}(u_1, u_2)\}; \Delta \vdash l{::}\Omega \equiv (\mathsf{Arrow}(u_1{::}\Omega, u_2{::}\Omega){::}\Omega) :: \Omega$$

$$\Phi \uplus \{l{::}\kappa{=}\lambda t{::}\kappa'.\mu\}; \Delta \vdash l{::}\kappa \equiv ((\lambda t{::}\kappa'.\mu){::}\kappa) :: \kappa$$

I claim that orienting these equivalences to the right yields a reduction system for constructors (and hence types) that is both locally confluent and strongly normalizing if Φ has no cycles, which is true when Φ is well-formed. It should be fairly straightforward to extend the proofs of Chapter 4 to show that this claim is true.

Judgment 11, program formation, is determined by the following rule:

$$\frac{\emptyset \vdash \Phi \qquad \Phi; \emptyset \vdash H : \Psi \qquad \Phi; \Psi; \emptyset; \emptyset \vdash e : \sigma}{\vdash (Q_\Phi, H, e) : \sigma}$$

The rule requires that the constructor heap be well-formed and described by Φ, that the expression heap be well-formed and described by Ψ under the assumptions of Φ, and that the expression be well-formed with type σ under the assumptions Φ and Ψ. Note that all components must be closed with respect to type and value variables.

Given unique normal forms for types, it is straightforward to define a suitable notion of normal derivation for λ_i^{ML}-GC programs and hence show that type checking is decidable. With the normal derivations in hand, it should be fairly straightforward to prove both preservation and progress, as in Chapter 4.

Proposition 7.5.1 (Preservation) *If* $\vdash (Q, H, e) : \sigma$ *and* $(Q, H, e) \longmapsto (Q', H', e')$, *then* $\vdash (Q', H', e') : \sigma$.

Proposition 7.5.2 (Progress) *If* $\vdash P : \sigma$, *then either* P *is an answer or else there exists a* P' *such that* $P \longmapsto P'$.

7.5.3 Garbage Collection and λ_i^{ML}-GC

The definitions of garbage and garbage collection for λ_i^{ML}-GC are essentially the same as for Mono-GC, except that I want to eliminate garbage bindings in both the constructor and the expression heaps. Assuming FL calculates all of the free locations of a program, then the tracing collection specification for λ_i^{ML}-GC is simply:

$$\frac{FL(Q_1, H_1, e) = \emptyset}{(Q_1 \uplus Q_2, H_1 \uplus H_2, e) \overset{\text{trace}}{\longmapsto} (Q_1, H_1, e)}$$

It is easy to prove the diamond and postponement lemmas hold with respect to this trace step and any other rewriting rule of λ_i^{ML}-GC. From this, we can conclude that the tracing collection specification is a sound garbage collection rule.

In the rest of this section, I will present a type-based, tag-free collection algorithm and give a proof sketch that it is sound by showing that any garbage it collects can also be collected by the tracing specification.

Before giving the collection algorithm, I need to define some auxiliary functions that extract the range of the constructor and value environments of an abstraction. Since λ_i^{ML}-GC uses meta-level substitution, these environments are not immediately apparent. Therefore, these auxiliary functions must deconstruct terms based on abstract syntax to find these values. In a lower-level model, as with Mono-GC, where environments are explicit, no such processing of terms is required.

The ConEnv function maps a constructor μ to a location kind assignment Π, by extracting all of the locations (and their kinds) within the constructor:

$$\begin{array}{lcl}
\mathsf{ConEnv}(t :: \kappa) & = & \emptyset \\
\mathsf{ConEnv}(\mathsf{Int} :: \kappa) & = & \emptyset \\
\mathsf{ConEnv}(l :: \kappa) & = & \{l{::}\kappa\} \\
\mathsf{ConEnv}(\mathsf{Arrow}(\mu_1, \mu_2) :: \kappa) & = & \mathsf{ConEnv}(\mu_1) \cup \mathsf{ConEnv}(\mu_2) \\
\mathsf{ConEnv}((\lambda t{::}\kappa_1.\mu) :: \kappa) & = & \mathsf{ConEnv}(\mu) \\
\mathsf{ConEnv}((\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_i; \mu_a)) :: \kappa) & = & \mathsf{ConEnv}(\mu) \cup \mathsf{ConEnv}(\mu_i) \cup \mathsf{ConEnv}(\mu_a)
\end{array}$$

The $\mathsf{TypeEnv}$ function maps a type σ to a location kind assignment Π:

$$\begin{array}{lcl}
\mathsf{TypeEnv}(T(\mu)) & = & \mathsf{ConEnv}(\mu) \\
\mathsf{TypeEnv}(\mathsf{int}) & = & \emptyset \\
\mathsf{TypeEnv}(\sigma_1 \to \sigma_2) & = & \mathsf{TypeEnv}(\sigma_1) \cup \mathsf{TypeEnv}(\sigma_2) \\
\mathsf{TypeEnv}(\forall t{::}\kappa.\sigma) & = & \mathsf{TypeEnv}(\sigma)
\end{array}$$

Finally, the ExpEnv function maps an expression e to both a location kind assignment Π and a location type assignment Ψ. In the definition of the function, I use $\langle \Pi_1, \Psi_1 \rangle \cup$

$\langle \Pi_2, \Psi_2 \rangle$ to abbreviate $\langle \Pi_1 \cup \Pi_2, \Psi_1 \cup \Psi_2 \rangle$.

$$
\begin{array}{lll}
\mathsf{ExpEnv}(x) & = & \langle \emptyset, \emptyset \rangle \\
\mathsf{ExpEnv}(v{:}\mathsf{int}) & = & \langle \emptyset, \emptyset \rangle \\
\mathsf{ExpEnv}(l{:}\sigma_1 \to \sigma_2) & = & \langle \mathsf{TypeEnv}(\sigma_1 \to \sigma_2), \{l{:}\sigma_1 \to \sigma_2\} \rangle \\
\mathsf{ExpEnv}(l{:}\forall t{::}\kappa.\sigma) & = & \langle \mathsf{TypeEnv}(\sigma), \{l{:}\forall t{::}\kappa.\sigma\} \rangle \\
\mathsf{ExpEnv}((\lambda x{:}\sigma'.e) : \sigma) & = & \langle \mathsf{TypeEnv}(\sigma) \cup \mathsf{TypeEnv}(\sigma'), \emptyset \rangle \cup \mathsf{ExpEnv}(e) \\
\mathsf{ExpEnv}((e_1\, e_2) : \sigma) & = & \langle \mathsf{TypeEnv}(\sigma), \emptyset \rangle \cup \mathsf{ExpEnv}(e_1) \cup \mathsf{ExpEnv}(e_2) \\
\mathsf{ExpEnv}((\Lambda t{::}\kappa.e) : \sigma) & = & \langle \mathsf{TypeEnv}(\sigma), \emptyset \rangle \cup \mathsf{ExpEnv}(e) \\
\mathsf{ExpEnv}((e_1\, [\mu]) : \sigma) & = & \langle \mathsf{TypeEnv}(\sigma) \cup \mathsf{ConEnv}(\mu), \emptyset \rangle \cup \mathsf{ExpEnv}(e) \\
\multicolumn{3}{l}{\mathsf{ExpEnv}((\texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma'](e_i; e_a)) : \sigma) =} \\
\multicolumn{3}{l}{\quad \langle \mathsf{TypeEnv}(\sigma) \cup \mathsf{TypeEnv}(\sigma') \cup \mathsf{ConEnv}(\mu), \emptyset \rangle \cup \mathsf{ExpEnv}(e_i) \cup \mathsf{ExpEnv}(e_a)}
\end{array}
$$

Note that, once a small value is reached, we can use type information to determine the shape of the value. For instance, when processing a small value v labelled with int, we know that v must be an integer i. Hence, we continue to use type information to determine the shape of small values and heap values, just as I did in the tag-free collection of Mono-GC.

The following lemma shows that the ConEnv and $\mathsf{TypeEnv}$ functions extract appropriate kind assignments from well-formed constructors and types.

Lemma 7.5.3

1. *If* $\Pi; \Delta \vdash (\nu :: \kappa)$, *then* $\mathsf{ConEnv}(\nu :: \kappa) = \Pi'$ *for some* Π' *and* $\Pi' \subseteq \Pi$ *and* $\Pi'; \Delta \vdash (\nu :: \kappa)$.

2. *If* $\Pi; \Delta \vdash \sigma$, *then* $\mathsf{TypeEnv}(\sigma) = \Pi'$ *for some* Π' *and* $\Pi' \subseteq \Pi$ *and* $\Pi'; \Delta \vdash \sigma$.

Proof (sketch): Simple induction on ν and σ, using the syntax-directedness of the formation rules. The fact that $\Pi' \subseteq \Pi$ ensures that the inductive hypotheses can be unioned to form a consistent kind assignment. $\Box$

The next lemma shows that we can strengthen the assumptions regarding the equivalence of two constructors or two types, as long as the strengthened assumptions are closed and cover the free locations of the constructors or types.

Lemma 7.5.4

1. *If* $\Phi; \Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$, $\Phi' \subseteq \Phi$, $\emptyset \vdash \Phi'$, $\Pi_{\Phi'}; \Delta \vdash \mu_1$, *and* $\Pi_{\Phi'}; \Delta \vdash \mu_2$, *then* $\Phi'; \Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$.

2. *If* $\Phi; \Delta \vdash \sigma_1 \equiv \sigma_2$, $\Phi' \subseteq \Phi$, $\emptyset \vdash \Phi'$, $\Pi_{\Phi'}; \Delta \vdash \sigma_1$, *and* $\Pi_{\Phi'}; \Delta \vdash \sigma_2$, *then* $\Phi'; \Delta \vdash \sigma_1 \equiv \sigma_2 :: \kappa$.

Proof (sketch): By induction on the derivation of $\Phi; \Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$ and $\Phi; \Delta \vdash \sigma_1 \equiv \sigma_2$. □

Next, I argue that if e has type σ under $\Phi; \Psi; \Delta; \Gamma$, then $\mathsf{ExpEnv}(e)$ exists and is some $\langle \Pi', \Psi' \rangle$ "consistent" with Φ and Ψ. Here, consistent means that, if we start with a subset of Φ containing all of the locations bound in Π', extend this subset so that it is closed and covers all of the free locations in Γ and σ yielding Φ', then $\Phi'; \Psi'; \Delta; \Gamma$ is sufficient to show that e has type σ.

Lemma 7.5.5 *If* $\Phi; \Psi; \Delta; \Gamma \vdash e : \sigma$, *then* $\mathsf{ExpEnv}(e) = \langle \Pi', \Psi' \rangle$ *and:*

1. $\Pi' \subseteq \Pi_\Phi$,
2. $\Psi' \subseteq \Psi$,
3. *If* $\Phi' \subseteq \Phi$, *such that* $Dom(\Pi') \subseteq Dom(\Phi')$, $\Pi_\Phi \vdash \Psi'$, $\Pi_\Phi; \Delta \vdash \Gamma$, $\Pi_\Phi; \Delta \vdash \sigma$, *and* $\vdash \Phi'$, *then* $\Phi'; \Psi'; \Delta; \Gamma \vdash e : \sigma$.

Proof (sketch): The proof proceeds by induction on e and relies upon the fact that each typing derivation has a normal form that interleaves non-equiv rules with equiv rules. The preconditions of part 3 coupled with the previous lemma are sufficient to show that, for every application of the equiv step, the two types in question are equivalent under the strengthened assumptions. Parts 1 and 2 are needed to show that the union of the assumptions for the inductive hypotheses are well-formed contexts. □

Using ConEnv, $\mathsf{TypeEnv}$, and ExpEnv, I can now define functions to extract the locations and their types or kinds from heap values based on kinds or types. The definitions of these functions are similar to the various TL functions of Mono-GC, given in Section 7.3. The function KL is a partial function that takes a kind κ and a constructor heap value q and returns the set of free locations in q as well as their kinds.

$$\begin{array}{rcl} KL[\Omega](\mathsf{Arrow}(u_1, u_2)) & = & \mathsf{ConEnv}(u_1{::}\Omega) \cup \mathsf{ConEnv}(u_2{::}\Omega) \\ KL[\kappa_1 \to \kappa_2](\lambda t{::}\kappa_1.\mu) & = & \mathsf{ConEnv}(\lambda t{::}\kappa_1.\mu) \end{array}$$

Similarly, the function TL is a partial function that takes a head-normal type ς and a heap value h and returns the set of free locations in h as well as their kinds or types.

$$\begin{array}{rcl} TL[\sigma_1 \to \sigma_2](\lambda x{:}\sigma'.e) & = & \mathsf{ExpEnv}((\lambda x{:}\sigma'.e) : \sigma_1 \to \sigma_2) \\ TL[\forall t{::}\kappa.\sigma](\Lambda t{::}\kappa.e) & = & \mathsf{ExpEnv}((\Lambda t{::}\kappa.e) : \forall t{::}\kappa.\sigma) \end{array}$$

As for Mono-GC, I express garbage collection as a set of rewriting rules between tuples. I begin by defining a rewriting system that only operates on constructor heaps. For this system, tuples are of the form (Q_f, Π_s, Q_t) where Q_f is the constructor from-space, Π_s

is the scan-set describing the kinds of all constructors in the from-space reachable from the to-space, and Q_t is the to-space. The rewriting rule for constructors is simply:

$$(Q_f \uplus \{l{=}q\}, \Pi_s \uplus \{l{::}\kappa\}, Q_t) \Rightarrow (Q_f, \Pi_s \cup \Pi'_s, Q_t \uplus \{l{=}q\})$$

$$\text{where } \Pi'_s = \{l'{::}\kappa' \in KL[\kappa](q) \mid l' \notin \mathit{Dom}(Q_t) \uplus \{l\}\}$$

If a location l, described by κ is in the scan-set and l is bound to the heap value q in the from-space, then we forward the binding $l{=}q$ to the to-space and add any free location in q and its kind to the scan-set, unless the location has already been forwarded to the to-space.

The following definition gives the essential invariants of the constructor garbage collection rewriting system:

Definition 7.5.6 (Constructor GC Well-Formedness) (Q_f, Π_s, Q_t) *is well-formed with respect to* Φ *iff:*

1. $\emptyset \vdash Q_f \uplus Q_t :: \Phi$,
2. $\Pi_s \subseteq \Pi_\Phi$,
3. $\mathit{Dom}(\Pi_s) \subseteq \mathit{Dom}(Q_f)$,
4. $\Pi_s \vdash Q_t :: \Pi_t$, *where* $\Pi_t = \{l{::}\kappa \in \Pi_\Phi \mid l \in \mathit{Dom}(Q_t)\}$.

It is straightforward to prove that constructor well-formedness is preserved by the rewriting system and that progress is always possible for well-formed tuples.

Lemma 7.5.7 (Constructor GC Preservation) *If* T *is well-formed with respect to* Φ *and* $T \Rightarrow T'$, *then* T' *is well-formed with respect to* Φ.

Proof (sketch): Follows from the invariants and lemma 7.5.3 (1). The argument is similar to the preservation argument for Mono-GC (see lemma 7.3.3). □

Lemma 7.5.8 (Constructor GC Progress) *If* $T = (Q_f, \Pi_s, Q_t)$ *is well-formed with respect to* Φ, *then either* Π_s *is empty or else there exists a* T' *such that* $T \Rightarrow T'$.

Proof: Suppose $\Pi_s = \Pi'_s \uplus \{l{::}\kappa\}$. By the second condition of well-formedness, we know that $\Pi_\Phi(l) = \kappa$. By the third condition, we know that $Q_f = Q'_f \uplus \{l{=}q\}$ for some Q'_f and q. By the first condition, we know that $\Pi_\Phi \vdash q{::}\kappa$. Hence, $KL[\kappa](q)$ is defined and by lemma 7.5.3, $KL[\kappa](q) \subseteq \Pi_\Phi$. Thus, $\Pi'_s \cup \Pi''_s$ is well-formed where $\Pi''_s = \{l'{::}\kappa' \in KL[\kappa](q) \mid l' \notin \mathit{Dom}(Q_t) \uplus \{l\}\}$. Therefore, $T \Rightarrow (Q_f, \Pi'_s \cup \Pi''_s, Q_t \uplus \{l{=}q\})$. □

Since the size of the from-space always decreases with each step, it is easy to see that constructor garbage collection always terminates. The progress lemma tells us that the collection never gets stuck and the preservation lemma tells us that at every step, the resulting tuple is well-formed with respect to the given Φ.

The following lemma shows that constructor garbage collection is locally confluent. With the fact that constructor garbage collection always terminates, this implies that constructor collection is confluent. This tells us that, no matter what order we process the constructors, we always get the same to-space at the end of a collection.

Lemma 7.5.9 (Constructor GC Local Confluence) *If T is well-formed with respect to Φ, $T \Rightarrow T_1$ and $T \Rightarrow T_2$, then there exists a T' such that $T_1 \Rightarrow T'$ and $T_2 \Rightarrow T'$.*

Proof: Suppose $T = (Q_f \uplus \{l_1{=}q_1, l_2{=}q_2\}, \Pi_s \uplus \{l_1{::}\kappa_1, l_2{::}\kappa_2\}, Q_t)$, $T_1 = (Q_f \uplus \{l_2{=}q_2\}, \Pi_s \uplus \{l_2{::}\kappa_2\} \cup \Pi_1, Q_t \uplus \{l_2{=}q_2\})$ where $\Pi_1 = \{l'{::}\kappa' \in KL[\kappa_1](q_1) \mid l' \notin Dom(Q_t) \uplus \{l_1\}\}$, and $T_2 = (Q_f \uplus \{l_1{=}q_1\}, \Pi_s \uplus \{l_1{::}\kappa_1\} \cup \Pi_2, Q_t \uplus \{l_2{=}q_2\})$ where $\Pi_2 = \{l'{::}\kappa' \in KL[\kappa_2](q_2) \mid l' \notin Dom(Q_t) \uplus \{l_2\}\}$. Let $T' = (Q_f, \Pi_s \cup \Pi'_1 \cup \Pi'_2, Q_t \uplus \{l_1{=}\ 1q_1, l_2{=}q_2\})$, where $\Pi'_1 = \{l'{::}\kappa' \in \Pi_1 \mid l' \neq l_2\}$ and $\Pi'_2 = \{l'{::}\kappa' \in \Pi_2 \mid l' \neq l_1\}$. Then it is easy to see that both $T_1 \Rightarrow T'$ and $T_2 \Rightarrow T'$. $\square$

Lemma 7.5.10 *If (Q_f, Π_s, Q_t) and $(Q_f, \Pi_s \cup \Pi'_s, Q_t)$ are well-formed with respect to Φ, then $(Q_f, \Pi_s, Q_t) \Rightarrow^* (Q'_f, \emptyset, Q'_t)$, $(Q_f, \Pi_s \cup \Pi'_s, Q_t) \Rightarrow^* (Q''_f, \emptyset, Q''_t)$, and $Q'_t \subseteq Q''_t$.*

For expressions, the garbage collection rewriting rules operate on 6-tuples of the form $(Q_f, H_f, \Pi_s, \Psi_s, Q_t, H_t)$, where Q_f and H_f are the constructor and expression from-spaces, Π_s describes the constructors immediately reachable from Q_t, H_t, and Ψ_s, and Ψ_s describes the heap values immediately reachable from H_t. There are two rewriting rules at this level. The first rule simply uses the constructor rewriting rule to process a constructor binding:

$$\frac{(Q_f, \Pi_s, Q_t) \Rightarrow (Q'_f, \Pi'_s, Q'_t)}{(Q_f, H_f, \Pi_s, \Psi_s, Q_t, H_t) \Rightarrow (Q'_f, H_f, \Pi'_s, \Psi_s, Q'_t, H_t)}$$

The second rule processes a binding in the expression heap:

$$(Q_f, H_f \uplus \{l{=}h\}, \Pi_s, \Psi_s \uplus \{l{:}\varsigma\}, Q_t, H_t) \Rightarrow (Q_f, H_f, \Pi_s \cup \Pi'_s, \Psi_s \cup \Psi'_s, Q_t, H_t \uplus \{l{=}h\})$$

$$\begin{aligned} \text{where } & \langle \Pi''_s, \Psi''_s \rangle = TL[\varsigma](h) \\ & \Pi'_s = \{l'{::}\kappa' \in \Pi''_s \mid l' \notin Dom(Q_t)\} \\ & \Psi'_s = \{l'{:}\varsigma' \in \Psi''_s \mid l' \notin Dom(H_t) \uplus \{l\}\} \end{aligned}$$

Note that both of the scan sets are updated when an expression heap value is processed.

The initialization and finalization steps for the full garbage collection are captured by the following inference rule:

$$\frac{\mathsf{ExpEnv}(e) = \langle \Pi, \Psi \rangle \qquad (Q, H, \Pi, \Psi, \emptyset, \emptyset) \Rightarrow^* (Q_f, H_f, \emptyset, \emptyset, Q_t, H_t)}{(Q, H, e) \overset{\text{tr-alg}}{\longmapsto} (Q_t, H_t, e)}$$

We initialize the system by extracting the locations and their kinds or types from the range of the environment of the current expression. This corresponds to extracting the root locations from the stack and registers of a real implementation. Then, we continue choosing locations in one of the scan sets, forward this location from the appropriate from-space to the appropriate to-space, potentially adding new locations to the scan-sets. Once the scan-sets become empty, the algorithm is finished, and the to-spaces are taken as the new, garbage collected heaps of the program.

To prove the correctness of the algorithm, I must establish a suitable set of invariants that guarantees that (a) the algorithm does not become stuck and (b) the resulting program is closed. The key difficulty in establishing the invariants is that we must conceptually complete the constructor garbage collection to ensure that enough constructors are present that we can derive all needed equivalences to type-check the heap and expression of the program.

Definition 7.5.11 (Well-Formedness) *Suppose* $P = (Q, H, e)$ *where* $\emptyset \vdash \Phi$, $\Phi; \emptyset \vdash H : \Psi$, *and* $\Phi; \Psi; \emptyset; \emptyset \vdash e : \sigma$. *The tuple* $T = (Q_f, H_f, \Pi_s, \Psi_s, Q_t, H_t)$ *is* well-formed *with respect to* P, Φ, Ψ, *and* σ *iff:*

1. (Q_f, Π_s, Q_t) *is well-formed with respect to* Φ *and thus, for some* $\Phi' \subseteq \Phi$, $(Q_f, \Pi_s, Q_t) \Rightarrow^* (Q'_f, \emptyset, Q_{\Phi'})$,
2. $H = H_f \uplus H_t$,
3. $\Psi_s \subseteq \Psi$ *and* $Dom(\Psi_s) \subseteq Dom(H_f)$,
4. $\Phi'; \Psi_s \vdash H_t : \Psi_t$ *where* $\Psi_t = \{l{:}\varsigma \in \Psi \mid l \in Dom(H_t)\}$, *and*
5. $\Phi'; \Psi_s \uplus \Psi_t; \emptyset; \emptyset \vdash e : \sigma$.

Roughly speaking, the invariants guarantee that (1) constructor garbage collection can proceed to some appropriate final state, (2) all of the values in the expression heap are accounted for, (3) the scan-set is consistent with the global location type assignment Ψ and describes a frontier of locations bound in the from-space, (4) after constructor collection is complete, the resulting constructor to-space and expression scan-set cover the free locations in the to-space and (5) all of the free locations in e and σ are covered by the to-spaces or scan-sets.

The following lemma shows that the invariants are strong enough to guarantee that, at the end of rewriting, the resulting program is a collection of the original program.

Lemma 7.5.12 *Let $P = (Q, H, e)$ where $\emptyset \vdash \Phi$, $\Phi; \emptyset \vdash H : \Psi$, and $\Phi; \Psi; \emptyset; \emptyset \vdash e : \sigma$, and suppose $T = (Q_f, H_f, \emptyset, \emptyset, Q_t, H_t)$ is well-formed with respect to P, Φ, Ψ, and σ. Then (Q_t, H_t, e) is a collection of P.*

Proof: From the correctness of the tracing collection specification, it suffices to show that (Q_t, H_t, e) is closed. Since T is well-formed, we know from the first invariant that $(Q_f, \emptyset, Q_t)$ is well-formed with respect to Φ. Thus, taking $\Phi' \subseteq \Phi$ such that $Dom(\Phi') = Dom(Q_t)$, we know that $\emptyset \vdash Q_t{::}\Pi_{\Phi'}$ and thus $\emptyset \vdash \Phi'$. From the fourth invariant, we know that $\Phi'; \emptyset \vdash H_t{:}\Psi_t$ where $\Psi_t = \{l{:}\varsigma \in \Psi \mid l \in Dom(H_t)\}$ and from the fifth invariant, $\Phi'; \Psi_t; \emptyset; \emptyset \vdash e : \sigma$. Consequently, $\vdash (Q_t, H_t, e) : \sigma$ and (Q_t, H_t, e) is closed. Therefore, $P \overset{\text{trace}}{\longmapsto} (Q_t, H_t, e)$ and thus $P \simeq (Q_t, H_t, e)$. □

Since either the size of the constructor from-space or the expression from-space strictly decreases at each step, it is clear that the rewriting system either terminates or gets stuck.

Lemma 7.5.13 (Preservation) *If T is well-formed with respect to P, Φ, Ψ, and σ, and $T \Rightarrow T'$, then T' is well-formed with respect to P, Φ, Ψ, and σ.*

Proof: In the first case, $T \Rightarrow T'$ via the constructor garbage collection rule. Well-formedness of T' is guaranteed by preservation of the constructor GC invariants, with confluence of constructor GC.

In the second case, $T = (Q_f, H_f \uplus \{l{=}h\}, \Pi_s, \Psi_s \uplus \{l{:}\varsigma\}, Q_t, H_t)$ and $T' = (Q_f, H_f, \Pi_s \cup \Pi'_s, \Psi_s \cup \Psi'_s, Q_t, H_t \uplus \{l{=}h\})$, where $TL[\varsigma](h) = \langle \Pi''_s, \Psi''_s \rangle$, $\Pi'_s = \{l'{::}\kappa' \in \Pi''_s \mid l' \notin Dom(Q_t)\}$, and $\Psi'_s = \{l'{:}\varsigma' \in \Psi''_s \mid l' \notin Dom(H_t) \uplus \{l\}\}$. Lemma 7.5.5 and the definition of TL tells us that $\Pi''_s \subseteq \Pi_s$ and $\Psi''_s \subseteq \Psi_s$. Thus $\Pi_s \cup \Pi'_s$ is a well-formed location kind assignment that is a subset of Π. Thus, invariant (1), $(Q_f, \Pi_s \cup \Pi'_s, Q_t)$ is satisfied. Invariant (2) is trivially satisfied, since, by assumption, $H = H_f \uplus H_t \uplus \{l{=}h\}$. Invariant (3) is satisfied if $\Psi'_s \subseteq \Psi$ and $Dom(\Psi') \subseteq Dom(H_f)$. The former condition holds since $\Psi''_s \subseteq \Psi$ and the latter condition holds by construction of Ψ'.

By invariant (1), $(Q_f, \Pi_s, Q_t) \Rightarrow^* (Q'_f, \emptyset, Q'_t)$ for some Q'_f and Q'_t, where $\Phi' \subseteq \Phi$ and $Q_{\Phi'} = Q'_t$. Furthermore, $(Q_f, \Pi_s \cup \Pi'_s, Q_t) \Rightarrow^* (Q''_f, \emptyset, Q''_t)$ and taking $\Phi'' \subseteq \Phi$ such that $Q_{\Phi''} = Q''_t$, and by lemma 7.5.10, we know that $\Phi' \subseteq \Phi''$. By invariant (3), $\Phi'; \Psi_s \uplus \{l{:}\varsigma\} \vdash H_t : \Psi_t$. Since $\Phi' \subseteq \Phi''$, $\Phi''; \Psi_s \uplus \{l{:}\varsigma\} \vdash H_t : \Psi_t$. By lemma 7.5.5, we know that $\Phi''; (\Psi_s \uplus \Psi_t) \cup \Psi'_s; \emptyset; \emptyset \vdash h : \varsigma$. Thus, $\Phi''; \Psi_s \cup \Psi'_s \vdash H_t \uplus \{l{=}h\} : \Psi_t \uplus \{l{:}\varsigma\}$ and invariant (4) is satisfied.

Finally, by invariant (5), $\Phi'; \Psi_s \uplus \Psi_t \uplus \{l{:}\varsigma\}; \emptyset; \emptyset \vdash e : \sigma$. Thus, $\Phi''; (\Psi_s \uplus \Psi_t \uplus \{l{:}\varsigma\}) \cup \Pi'_s; \emptyset; \emptyset \vdash e : \sigma$ and invariant (5) is satisfied. □

Lemma 7.5.14 (Progress) *If* $T = (Q_f, H_f, \Pi_s, \Psi_s, Q_t, H_t)$ *is well-formed with respect to* P*,* Φ*,* Ψ*, and* σ*, then either* Π_s *and* Ψ_s *are empty or else there exists a* T' *such that* $T \Rightarrow T'$*.*

Proof: If Π_s is non-empty, then progress is guaranteed by constructor GC progress. If the expression scan-set is non empty, then it is of the form $\Psi_s \uplus \{l{:}\varsigma\}$. By invariant (3), $\{l{:}\varsigma\} \in \Psi$ and l must be bound in the from space. Thus, assume the from-space is of the form $H_f \uplus \{l{=}h\}$ for some H_f and h. By lemma 7.5.5 and the definition of TL, we know that $TL[\varsigma](h)$ is defined and is some $\langle \Pi''_s, \Psi''_s \rangle$ such that $\Pi''_s \subseteq \Pi$ and $\Psi''_s \subseteq \Psi$. Therefore, taking $\Pi'_s = \{l'{::}\kappa' \in \Pi''_s \mid l' \notin Dom(Q_t)\}$ and $\Psi'_s = \{l'{:}\varsigma' \in \Psi''_s \mid l' \notin Dom(H_t) \uplus \{l\}\}$, we know that $\Pi_s \cup \Pi'_s$ is well-formed and $\Psi_s \cup \Psi'_s$ is well-formed. Thus, $T \Rightarrow (Q_f, H_f, \Pi_s \cup \Pi'_s, \Psi_s \cup \Psi'_s, Q_t, H_t \uplus \{l{=}h\})$. □

Corollary 7.5.15 (Tracing Algorithm Correctness) *If* P *is well-typed, then there exists a* P' *such that* $P \stackrel{\text{tr-alg}}{\longmapsto} P'$ *and* $P \simeq P'$*.*

Proof: Follows immediately from lemma 7.5.12, Preservation, and Progress. □

7.6 Related Work

The literature on garbage collection in sequential programming languages *per se* contains few papers that attempt to provide a compact characterization of algorithms or correctness proofs. Demers et al. [34] give a model of memory parameterized by an abstract notion of a "points-to" relation. As a result, they can characterize *reachability*-based algorithms including mark-sweep, copying, generational, "conservative," and other sophisticated forms of garbage collection. However, their model is intentionally divorced from the programming language and cannot take advantage of any *semantic* properties of evaluation, such as type preservation. Consequently, their framework cannot model the type-based collectors I describe here. Nettles [98] provides a concrete specification of a copying garbage collection algorithm using the Larch specification language. My specification of the free-variable tracing algorithm is essentially a high-level, one-line description of his specification.

Hudak gives a denotational model that tracks reference counts for a first-order language [67]. He presents an abstraction of the model and gives an algorithm for computing approximations of reference counts statically. Chirimar, Gunter, and Riecke give a framework for proving invariants regarding memory management for a language with a *linear* type system [30]. Their low-level semantics specifies explicit memory management based on reference counting. Both Hudak and Chirimar et al. assume a weak approximation of garbage (reference counts). Barendsen and Smetsers give a Curry-like type system for functional languages extended with *uniqueness* information that guarantees an object is only "locally accessible" [16]. This provides a compiler enough information to determine when certain objects may be garbage collected or over-written.

Tolmach [119] built a type-recovery collector for a variant of SML that passes type information to polymorphic routines during execution. Aditya and Caro gave a type-recovery algorithm for an implementation of Id that is equivalent to type passing [5] and Aditya, Flood, and Hicks extended this work to garbage collection for Id [6]. In both collectors, bindings for type variables are accumulated in type environments as I propose here.

However, the type systems of these languages are considerably simpler that λ_i^{ML}. In particular, they only support instantiation of polytypes and not general forms of computation (e.g., function call and Typerec). Furthermore, neither of these implementations allowed terms to examine types for operations, such as polymorphic equality or dynamic argument flattening. Tolmach took advantage of these properties by delaying the computation of a type instantiation until this instantiation was needed during garbage collection. In essence, he represented types as closures – a pair consisting of a type environment and a term with free variables whose bindings could be found in the environment. His "lazy" strategy for type instantiation avoided constructing types that are unneeded outside garbage collection.

In contrast, for languages like λ_i^{ML}, I propose computing type information eagerly to ensure that no computation, and thus no allocation occurs during garbage collection. The TIL compiler uses a hybrid tag-free scheme to avoid constructing types for all values. TIL also performs various optimizations to share as many type computations as is possible. These and other "real-world" implementation issues are discussed in Chapter 8.

Over the past few years, a number of papers on inference-based collection in monomorphic [22, 129, 23] and polymorphic [8, 49, 50, 43] languages appeared in the literature. Appel [8] argued informally that "tag-free" collection is possible for polymorphic languages such as SML by a combination of recording information statically and performing what amounts to type inference during the collection process, though the connections between inference and collection were not made clear. Baker [14] recognized that Milner-style type inference can be used to prove that reachable objects can be safely collected, but did not give a formal account of this result. Goldberg and Gloger [50] recognized that it is not possible to reconstruct the concrete types of all reachable values in an implementation of an ML-style language that does not pass types to polymorphic routines. They gave an informal argument based on traversal of stack frames to show that such values are semantically garbage. Fradet [43] gave another argument based on Reynolds's abstraction/parametricity theorem [104].

The style of semantics I use here is closely related to the allocation semantics used in my previous work on garbage collection [96, 95], but is slightly lower-level. In particular, I use closures and environments to implement substitution. In this respect, the semantics is quite similar to the SECD [80] and CEK machines [40]. The primary difference between my approach and these machines is that I make the heap explicit, which enables me to define a suitable notion of garbage and garbage collection.

Chapter 8

The TIL/ML Compiler

TIL, which stands for *Typed Intermediate Languages*, is a batch compiler that translates a subset of Standard ML to DEC Alpha assembly language. Together with David Tarditi, Perry Cheng, and Chris Stone, I have constructed TIL to explore some of the practical issues of type-directed translation and dynamic type dispatch. In this chapter, I give an overview of the design and implementation of TIL, catalog its features and drawbacks, and compare it to Standard ML of New Jersey — one of the best SML compilers currently available.

Throughout this chapter, I use the plural (e.g., "we") when referring to Tarditi, Cheng, Stone, and me. I reserve the singular (e.g., "I") when referring only to myself.

8.1 Design Goals of TIL

In designing TIL, our primary goal was to make the common case fast, possibly at the expense of a less common case. For example, all functions in SML take one argument; multiple arguments are simulated by using a tuple as the argument. From previous studies [81, 110], we determined that most functions do not use the tuple argument except to extract the components of the tuple. Consequently, we wanted TIL to translate functions so that they take tuple components in registers as multiple arguments, thereby avoiding constructing the argument tuple.

Our secondary goal was to use type-directed translation to propagate type information through as many stages of compilation as was possible. The idea was to try to discover new ways that types could be used in the lower-levels of a compiler. Some uses of types came at a surprisingly low level. For instance, we used type information to orient switch arms to maximize correctly predicted branches for list and tree-processing code. To support tag-free garbage collection, we knew that it was necessary to hang on to as much type information for as long as was possible. To accomplish this, we needed a suitably expressive, typed intermediate form.

Another design goal was to leverage existing tools as much as possible. For instance, we decided to emit Alpha assembly language and let the native assembler handle instruction scheduling and opcode emission. We were careful to use standard Unix tools, such as `ld` so that we could take advantage of profilers, debuggers, and other widely used tools. Also, to avoid constructing a parser, type-checker, and pattern match compiler, we decided to use the front end of the ML Kit Compiler [19].

Finally, we wanted TIL to be as interoperable with other languages (notably C, C++, and Fortran) as possible, without compromising the efficiency of conventional SML code. The goal was to support efficient access to library routines, system calls, and hardware, which is needed for "systems" programming in SML as proposed by the Fox project [56]. To this end, we decided to use tag-free garbage collection to support untagged, unboxed integers and pointers, since most arguments to libraries or system calls involve these two representations. Also, we made the register allocator aware of the standard C calling convention so that C functions could be directly called from SML code.

We decided not to use a fully tag-free collector, but instead to place tag words before heap-allocated objects (i.e., records and arrays). For records, we decided to use a bit map to describe which components are pointers. Arrays are uniform, so we planned to use a single bit in the length tag to tell whether or not the contents of the array are pointers.

Most allocators for languages like C and C++ also use header words for heap-allocated objects, so the proposed scheme would not sacrifice interoperability. Furthermore, we suspected that most tags could be computed at compile time, so the cost of constructing these tags would not be prohibitive. We also felt that tagging heap objects would simplify the garbage collector since we could use standard breadth-first copying collection once the "roots" (i.e., registers and stack) had been scanned. Fully tag-free collectors cannot use the standard breadth-first scan, because there is not always space for a forwarding pointer in a tag-free object[1]. Furthermore, we were worried that the sizes of tables that contain full type information might be excessively large [35]. Taking all of these factors into account, we felt that leaving integers and pointers untagged, but tagging heap-allocated objects, had the most virtues.

We decided not to unbox double-precision floating point values except within functions and within arrays. We felt that unboxing doubles in arrays would be important for scientific code (e.g., matrix operations). Unboxing double function arguments requires a more complicated approach to calling conventions and register allocation in the presence of polymorphism. (I discuss this issue in Section 8.8). Similarly, unboxed floating point values in records require a more complicated mechanism for calculating the sizes and tags of records, as well as the offsets of fields within records. We were unsure whether the run

[1]Yasuhiko Minamide pointed this out to me and showed that Tolmach failed to properly correct for this in his tag-free collector [119].

time costs of the more complicated mechanisms would outweigh the benefits of unboxed doubles, especially for conventional SML code which typically manipulates many records and few doubles.

We did design the compiler so that unboxed doubles could either be passed as function arguments or placed in records for monomorphic code. We hope to use TIL in the future to explore the tradeoffs of various mechanisms that support unboxed doubles for polymorphic code.

8.2 Overview of TIL

Figure 8.1 gives a block-diagram of the stages of the TIL compiler. All of the transformations are written in Standard ML. In this section, I give a brief overview of each stage and in the following sections, I provide more detailed information.

The first phase of TIL uses the front end of the ML Kit compiler [19] to parse, type check, and elaborate SML source code. The Kit produces annotated abstract syntax for the full SML language and then compiles a subset of this abstract syntax to an explicitly-typed core language called Lambda. The compilation to Lambda eliminates pattern matching and various derived forms.

I extended Lambda to support signatures, structures (modules), and separate compilation. Each source module is compiled to a Lambda module with an explicit list of imported modules and their signatures. Imported signatures may include transparent definitions of types defined in other modules. Hence, TIL supports a limited form of *translucent* [58] or *manifest* types [83].

I extended the mapping from SML abstract syntax to Lambda so that SML structures are mapped to Lambda structures with transparent imported types. Currently, the mapping to Lambda does not handle source-level signatures, nested structures, or functors. In principle, however, all of these constructs are supported by the intermediate languages of TIL.

The next phase of TIL uses an intermediate language called Lmli. Lmli is a "real world" version of λ_i^{ML}, providing constructs for dynamic type dispatch, efficient data representations, recursive functions, arrays, and so forth. In the translation of Lambda to Lmli, we use these constructs to provide tag-free polymorphic equality, specialized arrays, efficient data representations, and multi-argument functions. The argument flattening and implementation of polymorphic equality are based on the formal type-directed translation of Chapter 5.

Like λ_i^{ML}, type checking Lmli terms is decidable. I provide a kind checker, constructor normalizer, and type checker for Lmli. We also provide support for pretty-printing Lmli terms. Currently, the type checker is quite slow because I normalize all types before comparing them. A much better approach is to compare types directly and normalize

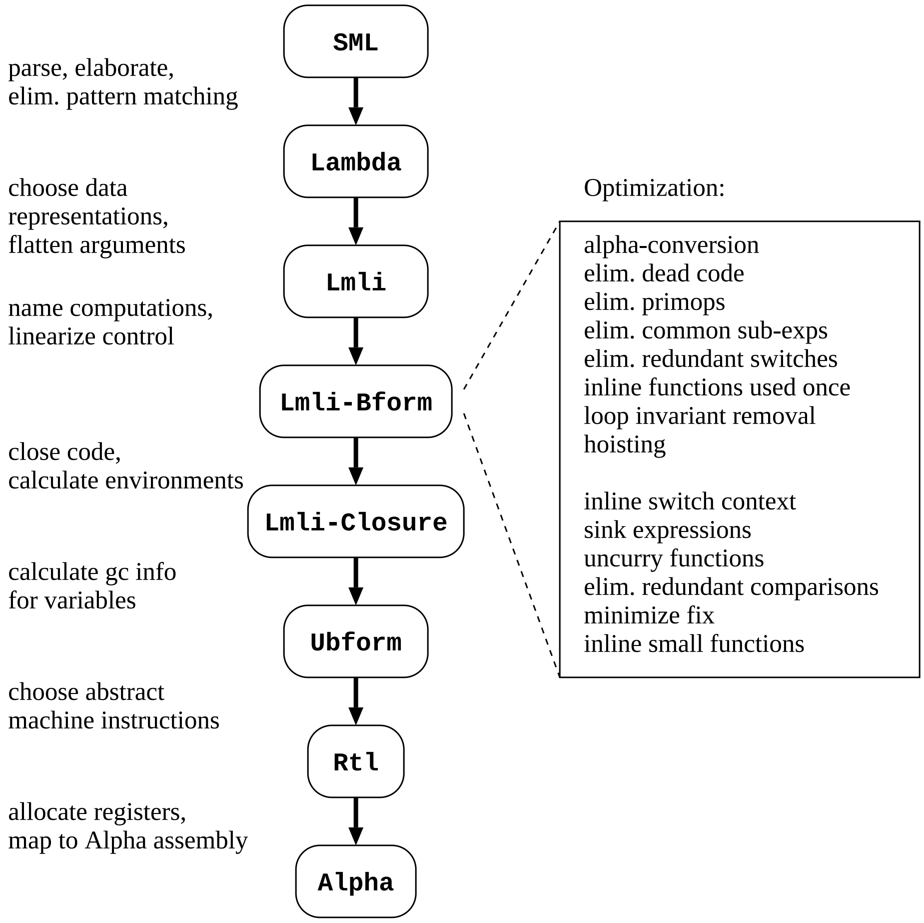


Figure 8.1: Stages in the TIL Compiler

components only if they do not match.

Lmli-Bform (or simply Bform) is a subset of Lmli similar to A-normal form [42]. It provides a more regular intermediate language than Lmli to facilitate optimization. Because Bform is a subset of Lmli, we can use all of the Lmli tools, including the type checker and pretty printer on the Bform representation[2]. We perform a wide variety of optimizations on the Bform representation of a program including dead code elimination; uncurrying; constant folding; constant typecase and switch elimination; inlining of constructor functions, term functions, type functions and switch continuations; common sub-expression elimination; redundant switch elimination; and invariant removal. Because the optimization phases use Bform for both the source and target language, the output of each phase can be checked for type correctness.

Most of the design and implementation of the optimizer is not my work, and is described fully by Tarditi's thesis [115]. It is interesting to note that working with a typed intermediate form did not constrain the set of optimizations that Tarditi wished to perform, and that types could be used to perform some optimizations that were not possible in an untyped setting. However, working with a typed intermediate form did have some drawbacks. In particular, our typed intermediate form needs more constructs (e.g., `typecase`) than a comparable untyped form. This makes the optimizer code bigger since there are more cases to process. In turn, this increases the likelihood of introducing bugs in compilation. However, we found that the ability to type-check the output of the optimizer often mitigated this drawback.

After Bform optimization, we perform closure conversion, mapping the Bform representation to a language called Lmli-Close. In fact, all of the constructs of Lmli-Close are present in Bform but are unused until the closure phase of the compiler. Hence, Lmli-Close is a refinement of Lmli-Bform, much the same as Lmli-Bform is a refinement of Lmli. The conversion is based on the type-directed closure translation described in Chapter 6. However, following Kranz [78], we calculate the set of functions that do not "escape" and avoid constructing closures for such functions. Because Lmli-Close is a subset of Bform, we can use both the optimizer and the type checker on the closure converted code.

After closure conversion, we translate the resulting code to an untyped Bform, called Ubform. Instead of annotating variables with types, Ubform requires that we annotate variables with representation information. The translation to Ubform erases the distinction between computations at the constructor and term levels. For example, a constructor function call looks the same as a term function call.

The next phase of TIL maps Ubform programs to the Rtl intermediate form. Rtl,

[2]The actual ML datatypes used for Lmli and Bform differ, but we provide a simple map from Bform to Lmli. If SML provided refinement types [44], then we could have defined Bform as a refinement of Lmli and avoided this extra piece of code.

which stands for register transfer language, provides an idealized RISC instruction set, with a few heavy-weight instructions and an infinite number of registers. After Rtl, we perform register allocation and map the resulting code to DEC Alpha assembly language.

We use the system assembler to translate Alpha assembly language to binaries and the system linker to link these binaries with the runtime system. The runtime is written in C and provides code for initialization, memory management, and multi-threading. The garbage collector uses table information generated at the Rtl level to determine which registers and stack slots contain pointer values. The rest of the garbage collector is a standard, two-space copying collector.

8.3 SML and Lambda

Currently, we do not support the signatures, nested structures, or functors of Standard ML. There are also some parts of core SML that we do not support. In order to assign a quantified, polymorphic type to an expression, we require that that expression be syntactically equivalent to a *value*. By value, we mean that the expression must be a constant, a record of values, a data constructor (besides `ref`) applied to values, or a function. This so-called "value restriction" is necessary to support a type-passing interpretation of SML, since polymorphic computations are represented as functions. The value restriction has been proposed by others [63, 57, 82] as a way to avoid the well-known problems of polymorphism and refs, exceptions, continuations, and other constructs that have computational effects. Furthermore, according to a study performed by Wright [131], most SML code naturally obeys the value restriction. The few cases he found that do not, are easily transformed so that they do meet this restriction.

The other restriction on core SML code involves datatypes. We do not support recursive datatypes of the form:

datatype α `foo = D1` **of** `(`α `*` α`) foo -> int`

where a type constructor `foo` abstracts a type argument α, and is defined in terms of itself applied to a *different* type containing the abstracted variable (e.g., `(`α `*` α`) foo`). Representing such a datatype as a predicative constructor is impossible because the type variable α must be abstracted inside the recursion equation governing the definition. Hence, the recursion equation defines a fixed-point over polytypes instead of monotypes. Such datatypes are very rare. A cursory study showed that no datatypes of this form existed in either the Edinburgh or the SML/NJ library. In many respects, the ability to define such datatypes violates the type-theoretic "essence of SML" [94], and thus I view them more as a bug in the Definition [90] than a feature. In principle, TIL supports the rest of the Standard ML Definition, though of course there may be bugs in the implementation.

We use the ML Kit Compiler [19] to parse, type-check, and elaborate SML expressions. The Kit translates SML to annotated abstract syntax. The annotations include position information for error reporting as well as type information. Next, the annotated abstract syntax is translated to the Lambda intermediate form. This intermediate form is quite similar to the language described by Birkedal et. al. [19], but I added support for type abbreviations, structures, and signatures. Also, we added various primitive operations to the language to support, for instance, unsigned integer operations, logical operations, and so forth.

The Kit compiler translates core SML, annotated abstract syntax to Lambda declarations. During the translation, all type definitions are hoisted to the top-level and almost all pattern matching is eliminated. Datatype definitions are represented in almost exactly the same fashion as they are at the source level.

I modified the translation to Lambda to support structures and a standard "prelude" environment. This environment contains bindings for commonly used functions such as `map` and `fold`, as well as definitions of the built in types, including arrays, bools, and strings. The prelude environment is conceptually prepended onto every structure. This allows the optimizer to easily inline functions such as `map`.

Some primitive types, notably strings, are defined in terms of other datatypes in the prelude. For example, the string type is represented as a standard datatype of the form:

```
datatype string_rep =
      C000 | C001 | C002 | ... | C255 |
      stringrep_str of int * (int array)
```

The data constructors `C000`—`C255` correspond to 8-bit, ASCII characters whereas the data constructor `stringrep_str` corresponds to strings of length 0 or of length greater than 1. We found that distinguishing characters from other strings was important since characters could always be represented unboxed and character comparison could be implemented efficiently[3]. Strings of more than one character are represented as integer arrays. Since integers are untagged, each integer in the array holds 4 characters on a 32-bit machine[4]. The extra `int` paired with the integer array indicates the length of the string in characters. All of the string primitives — including implode, explode, and append — are implemented by a combination of pattern matching and array operations. The ability to manipulate strings a word-at-a-time using the standard integer array operations simplified the implementation greatly without sacrificing performance.

The translation does provide support for separate compilation. In particular, each SML structure is compiled to a Lambda module containing a list of all imported modules

[3]Earlier versions of SML/NJ used a similar representation, but newer versions expose the character datatype to the programmer. Our string representation is compatible with either approach.

[4]Although the Alpha is a 64-bit processor, we represent integers and pointers as 32-bit values.

and their Lambda signatures. These signatures can, but need not, contain transparent or manifest type definitions. Since we account for all external references in the list of imported modules, each module can be compiled separately from other modules in the program. This scheme relies upon unique module names at link-time to resolve inter-module references.

8.4 Lmli

The Lmli intermediate form is based on the formal λ_i^{ML} calculus of Chapter 3, but provides a suitably rich set of constructs to support efficient compilation of Lambda datatypes and terms. In this section, I present various details regarding Lmli, including the SML datatype used to represent the kinds, constructors, types, and terms of Lmli.

8.4.1 Kinds, Constructors, and Types of Lmli

The SML datatype definitions for a subset of the kinds and constructors of Lmli are given in Figure 8.2. To simplify the presentation, I have eliminated all of the constructs that are used only for closure conversion. These constructs are discussed in Section 8.7.

All constructors are labelled with a kind. This simplifies kind checking and constructor manipulation, since we can always determine a constructor's kind with no additional context. A more sophisticated implementation might elide much of this information and reconstruct it as needed. The kind `Mono_k` is the ASCII representation of Ω and thus represents all constructors corresponding to monotypes. Kinds include `Mono_k`, n-ary products (`Tuple_k`), arrow kinds (`Arrow_k`), and list kinds (`List_k`). In TIL, we elide the distinction between a constructor μ of kind `Mono_k` and the type $T(\mu)$. Instead, we use the special kind `Poly_k` to distinguish types from constructors[5].

Constructors include variables, projections from modules (`Dot_c`), primitive constructors of zero or one argument, tuple intro and elim forms, list intro and elim forms, function intro and elim forms, recursive constructors (`Mu_c`), and monotype elim forms. I briefly discuss each of these constructs here, and then provide more details regarding primitive constructors.

The list intro forms include `Nil_c` and `Cons_c`. The list elim forms include both a fold for lists (`Fold_c`) and a simple case mechanism (`Listcase_c`). The formation rule

[5]TIL actually provides more kind structure than is shown here, in order to check well-formedness of types.

```
datatype kind = Mono_k | Tuple_k of kind list
  | Arrow_k of kind * kind | List_k of kind | Poly_k

datatype primcon = Primcon0 of primcon0 | Primcon1 of primcon1

datatype con = Con of (raw_con * kind)
and raw_con =
    Var_c of var
  | Dot_c of strid * int * label
  | Prim0_c of primcon0
  | Prim1_c of primcon1 * con
  | Tuple_c of con list
  | Proj_c of int * con
  | Nil_c of kind
  | Cons_c of con * con
  | Listcase_c of {arg : con, nil_c : con, cons_c : confn}
  | Fold_c of {arg : con, nil_c : con, cons_c : confn}
  | Fn_c of confn
  | App_c of con * con
  | Mu_c of ((var * con) list) * con
  | Typecase_c of {arg    : con,
                   arms   : (primcon * confn) list,
                   default: con,
                   kind   : kind}
  | Typerec_c of {arg     : con,
                  arms    : (primcon * confn) list,
                  default : con,
                  kind    : kind}
  | Let_c of (var * kind * con * con)
  | All_c of (var * kind) list * con
and confn = CF of (var * kind * con)
```

Figure 8.2: Kinds and Constructors of Lmli

for the fold construct is roughly

$$\frac{\Delta \vdash \mu :: \texttt{List\_k}(\kappa') \qquad \Delta \vdash \mu_1 :: \kappa \qquad \Delta \vdash \mu_2 :: \texttt{Tuple\_k}[\kappa', \texttt{List\_k}(\kappa'), \kappa] \to \kappa}{\Delta \vdash \texttt{Fold\_c}\{\texttt{arg=}\mu\texttt{,nil\_c=}\mu_1\texttt{,cons\_c=}\mu_2\} :: \kappa}$$

The `arg` component must be of list kind and thus evaluates to either a `Nil_c` or else a `Cons_c` constructor. The `nil_c` component is selected for the base case and the `arg_c` component is selected for the inductive case. The head and tail of a `Cons_c` cell are passed to the `cons_c` component as arguments, with the unrolling of the fold on the tail of the list. The formation rule for listcase is similar, but the `nil_c` component only takes the head and tail as arguments.

The monotype elim forms include both `Typerec_c` and `Typecase_c` forms. The clauses are indexed by a `primcon`, and each `primcon` can occur in at most one clause. Clauses corresponding to `primcon1` values are functions that take appropriate arguments of the appropriate kind, whereas clauses corresponding to `primcon0` values are constructor functions that take an empty tuple as an argument. The default case is used to match constructors that do not appear in the arms list. Note that although recursive types are considered to be monotypes, there is no way to deconstruct them in Lmli; the default clause of a `Typecase_c` or `Typerec_c` is always selected when one of these constructs is applied to a `Mu_c` constructor.

The function intro (`Fn_c`) and elim (`App_c`) forms are standard. The `Let_c` form can be abbreviated with these constructs as usual. However, this is not possible in the restricted Lmli-Bform that is used in the optimizer. We retain `Let_c` here to support pretty-printing, and both kind and type-checking of Lmli-Bform. The `All_c` constructor is not really a constructor, but rather a type. It is always labelled with the kind `Poly_k`.

The `Mu_c` constructor is a generalized recursive type constructor. Informally, `Mu_c` corresponds to a "letrec" at the constructor level, simultaneously binding the variables to the constructors, within the scope of the exported constructor. Each of the variables is constrained to have the kind `Mono_k`. We require that the constructors bound to the variables be *expansive*. That is, if one of the variables bound in the `Mu_c` occurs in a constructor bound to a variable, then that variable must occur within an argument to a `primcon1`.

The type that a recursive constructor represents is isomorphic to the type obtained by simultaneously replacing each variable in the exported constructor with the "unrolling" of the recursive type. For instance, the recursive constructor `Mu_c([(x1,c1),(x2,c2)],c)` is isomorphic to `{c1'/x1,c2'/x2}c` where

```
c1' = Con(Mu_c([(x1,c1),(x2,c2)],Var_c(x1,Mono_k)),Mono_k)
```

and

```
c2' = Con(Mu_c([(x1,c1),(x2,c2)],Var_c(x2,Mono_k)),Mono_k).
```

This isomorphism is not implicit as in some calculi. At the term level, we must use explicit "roll" and "unroll" operations on terms to affect the isomorphism.

The addition of recursive types to λ_i^{ML} is not straightforward since we have destroyed the key property that the monotypes can be generated by induction. However, the elim forms that we provide at the constructor level treat recursive types as pseudo-base cases. In particular, we cannot examine the contents of a `Mu_c` constructor with `Typecase_c` or `Typerec_c`. I therefore speculate that both confluence and strong-normalization of constructor reduction are preserved, though I have yet to prove this.

The `primcon0` and `primcon1` datatypes are defined as follows:

```
datatype primcon0 = Int_c | Real_c | String_c | Intarray_c
  | Realarray_c | Exn_c | Enum_c of int

datatype primcon1 = Ptrarray_c | Arrow_c | Sum_c | Sumcursor_c
  | Record_c | Recordcursor_c | Excon_c | Deexcon_c
  | Enumorrec_c of int | Enumorsum_c of int
```

Most of the `primcon0` data constructors are self explanatory. We provide a string constructor even though string values are represented in terms of other constructs (integer arrays). This distinction is necessary to support the proper semantics of polymorphic equality, since strings are compared by value whereas arrays are compared by reference. The `Exn_c` constructor is used to type exception packets, and the `Enum_c` constructor is used in the translation of datatypes. Enum values are used to represent data constructors with no argument (e.g., `nil`). Enum values are also used to tag variant records. We assume that enum values are always distinguishable from pointers to heap-allocated objects. Currently, we represent enum values as small integers between 0 and 255. We could represent enum values as odd integers, assuming pointers are always evenly aligned.

The `primcon1` data constructors are primitive constructors of one argument. Multiple arguments are simulated by a constructor tuple or a list of constructors. The `Ptrarray_c` (pointer array) constructor corresponds to arrays that contain any value except for integers or reals. The `Arrow_c` (arrow) constructor corresponds to functions at the term level. Functions in Lmli take multiple arguments and yield one result. Therefore, the arrow constructor takes two arguments (as a constructor tuple), and these arguments correspond to the domain types and the range type. The domain types are represented as a list of constructors. Thus, arrow has the kind `Tuple_k[List_k(Mono_k), Mono_k]` $\rightarrow$ `Mono_k`.

The `Record_c` (record) constructor corresponds to n-tuples at the term level and has the kind `List_k(Mono_k)` $\rightarrow$ `Mono_k`. `Recordcursor_c` values are used to iterate over the components of a `Record_c` value. Roughly speaking, a record cursor is a pair consisting of a pointer to a record and an integer offset.

The `Sum_c` (sum) constructor represents immutable, Pascal-style variant record types. Sum values are in fact records where the first component of the record contains an `Enum_c` value indicating the variant. Thus, the sum constructor has kind `List_k`(`List_k`(`Mono_k`)) $\rightarrow$ `Mono_k`. For instance, values with a type described by the constructor

```
Sum_c[[Int_c,Real_c],[String_c]]
```

are either records with a type described by

```
Record_c[Enum_c 2,Int_c,Real_c]
```

or else records with a type described by

```
Record_c[Enum_c 2,String_c].
```

Similar to record cursors, `Sumcursor_c` values provide a means for folding a computation across a sum.

Values described by `Excon_c` and `Deexcon_c` are used to represent SML exceptions. In particular, each creation of an SML exception carrying type τ is translated to a single operation that creates a pair consisting of an `Excon_c(`τ`)` value and a `Deexcon_c(`τ`)` value. A value of type `Excon_c(`τ`)` can be applied to a value of type τ to yield a value of type `Exn_c`, thereby hiding the type τ. A value of type `Deexcon_c`(τ) can be applied to a value of type `Exn_c`. The result is essentially a τ option: a variant record where the first variant is empty (`None`) and the second variant contains a value of type τ (`Some`).

Finally, the `Enumorrec_c` and `Enumorsum_c` constructors are special cases of variant records used to optimize the representation of SML datatypes. In particular, enum-or-rec values are either an enum value or a record value, whereas enum-or-sum values are either an enum value or a variant record (i.e., sum) value. Since records and sums are always allocated (i.e., pointers), we can always distinguish enum values from records and sums.

8.4.2 Terms of Lmli

The terms of Lmli are described by the SML datatype given in Figure 8.3. Similar to constructors, each term is labelled with its type, where the type is represented as an Lmli `con`. Labelling each term with its type simplifies type checking and type-directed translation. The space overheads of this fully typed representation are not quite as great as we might first expect. This is because we can bind types to variables (via `Let_e`) and use the variable in place of the type representation. Then, we can use standard optimization techniques, such as common sub-expression elimination and hoisting, to eliminate redundant type definitions.

```
datatype exp = Exp of (raw_exp * con)
and raw_exp =
    Var_e of var
  | Dot_e of strid * int * label
  | Record_e of exp list
  | Inject_e of int * (exp list)
  | Int_e of word
  | Real_e of string
  | Enum_e of int
  | String_e of string
  | Let_e of decl * exp
  | Fn_e of function
  | Tfn_e of tfunction
  | App_e of exp * (exp list)
  | Tapp_e of exp * (con list)
  | Coerce_e of coerceop * exp
  | Op1_e of op1 * exp
  | Op2_e of op2 * exp * exp
  | Misc_e of miscop
  | Switch_e of switch_exp
  | Typecase_e of typecase_exp
  | Tlistcase_e of tlistcase_exp
  | Raise_e of exp
  | Handle_e of exp * function
  | Export_e of
       {types : (label * con) list, values: (label * exp) list}
and decl =
    Var_d of (var * con * exp)
  | Con_d of (var * kind * con)
  | Fix_d of (var * con * function) list
  | Fixtype_d of (var * con * tfunction) list
and function = Func of (var * con) list * exp
and tfunction = Tfunc of (var * kind) list * exp
```

Figure 8.3: Terms of Lmli

The raw expressions include variables, projections from imported modules, literal values, various primitive operations, various switch operations, (recursive) value and constructor abstractions, value and constructor applications, exception primitives, `let`-expressions, and a mechanism for exporting type definitions and values. In the rest of this section, I briefly describe some of these constructs.

The `Inject_e` expression is used to calculate a sum value (i.e., variant record). The integer component must be an enum value (i.e., between 0 and 255). Operationally, `Inject_e` just allocates a record of size $n + 1$, places the enum component in the first position and then places the other values in positions 2 through $n + 1$.

`Let_e` expressions provide a means of declaring variables within a scope. Variables in a declaration are either bound to expressions (`Var_d`), constructors (`Con_d`), or mutually recursive functions. The `Fix_d` declaration provides fixed-points for value abstractions whereas the `Fixtype_d` declaration provides fixed-points for constructor abstractions. Both value and constructor abstractions can take multiple arguments.

Coercions are primitive operations that have no operational effect, but are needed for type checking. The set of coercion operations is given by

```
datatype coerceop =
    proll | punroll | penum_enumorrec | prec_enumorrec
  | penum_enumorsum | psum_enumorsum | penum2int | pfromstring
  | ptostring | pchr.
```

The `proll` and `punroll` coercions affect the isomorphism between a recursive type and its unrolling. The `penum_enumorrec`, and `penum_enumorsum` coercions inject an enum value into an enum-or-record or enum-or-sum type, whereas the `prec_enumorrec` and `psum_enumorrec` inject a record or sum into an enum-or-record or enum-or-sum type. The `penum2int` coercion coerces an enum to an integer value. The `pfromstring` and `ptostring` coerce a string to and from its underlying representation. Finally, the `pchr` operation coerces an integer to an enum value[6].

The `op1` primitive operations are given by the datatype

```
datatype op1 =
    preal_i | pnot_i | pfloor_r | psqrt_r | psin_r | pcos_r
  | parctan_r | pexp_r | pln_r | psize_a of spclarray
  | pselect of int | prec_cursor | prec_head | prec_tail
  | psum_cursor,
```

where

[6] Technically, the `pchr` operation should ensure that the integer value meets the representation constraints of enum values. In practice, the front-end ensures this by checking to see if the integer lies between 0 and 255.

```
datatype spclarray = Intarray | Realarray | Ptrarray.
```

Most of the rest of the operations perform some computation on real values, such as calculating the square root (`psqrt_r`) or natural log (`pln_r`). The `psize_a` operation calculates the size of an array.

The `prec_cursor` operation takes a record of type `Record_c[c1,...,cn]` and pairs it with the integer 0 to form a record cursor of type `Recordcursor_c[c1,...,cn]`. Similarly, the `psum_cursor` operation takes a sum value of type `Sum_c[c1,...,cn]` and pairs it with the enum 0 to form a sum cursor of type `Sumcursor_c[c1,...,cn]`. The `prec_head` operation takes a record cursor of type `Recordcursor_c[c1,...,cn]` and returns a value of type `c1`. This value is obtained by taking the current offset of the record cursor and selecting this component from the record of the record cursor. The `prec_tail` operation takes a record cursor of type `Recordcursor_c[c1,c2,...,cn]` and returns a new record cursor of type `Recordcursor_c[c2,...,cn]`. This new cursor is obtained by taking the offset of the old cursor, incrementing it, and then pairing it with the record of the old cursor to form a new cursor. When combined with the term-level listcase operations on constructors, record cursors can be used to fold an operation across the components of a record. We use this facility, for example, to compute polymorphic equality on records of arbitrary arity.

The `op2` primitive operations take two arguments and are defined as follows:

```
datatype op2 =
    pdiv_i | pmul_i | pplus_i | pminus_i | pmod_i
  | peq_i | plst_i | pgtt_i | plte_i | pgte_i
  | pdiv_ui | pmul_ui | pplus_ui | pminus_ui
  | plst_ui | pgtt_ui | plte_ui | pgte_ui
  | por_i | pand_i | pxor_i | plshift_i | prshift_i
  | pdiv_r | pmul_r | pplus_r | pminus_r | peq_r
  | plst_r | pgtt_r | plte_r | pgte_r
  | palloc_a of spclarray | psub_a of spclarray
  | pexcon | pde_excon | peqptr
```

Operations ending in "`_i`" are signed, checked integer operations, whereas operations ending in "`_ui`" are unsigned, unchecked operations. (The checks are for overflow and divide by zero.) The operations ending in "`_r`" are double-precision (i.e., 64-bit) IEEE floating point operations.

The `palloc_a` operations allocate arrays of the appropriate type, where the size is determined by the first argument and the array is initialized with the second argument. The `psub_a` operation extracts a value from an array (the first argument) at the given offset (the second argument). The `pexcon` operation takes a value of type `Excon_c(`τ`)` and a value of type τ and returns a value of type `Exn_c`. The `pde_excon` operation takes a value of type Exn_c and a value of type Deexcon_c(τ) and returns a Enumorrec_c(1,[τ]) value. The resulting value is either an enum or else a record containing a τ value.

The miscop operations are given by the following datatype:

```
datatype miscop =
    pupdate_a of spclarray * exp * exp * exp
  | pextern of string * con
  | pnew_exn of con
  | peq of con
  | vararg of con * exp
  | onearg of con * exp * exp
```

The pupdate_a operation is used to update an array. The front-end ensures that the index is within range. The pextern operation is used to reference external labels, exported by the runtime or a foreign language (e.g., C). The pnew_exn operation takes a constructor τ and returns a pair of a Excon_c(τ) value and a Deexcon_c(τ) value.

The peq operation corresponds to polymorphic equality at the monotype denoted by the given constructor. This operation can be coded using various other operations in the language (see Section 5.2.3). In fact, a later stage in the compiler replaces all occurrences of peq with a call to such a function defined in a separate, globally shared module. We leave the operation as a primitive so that the optimizer can easily recognize and specially treat the operation.

The vararg(τ,e) and onearg(τ,e,e') operations are used to implement dynamic argument flattening and roughly correspond to the vararg and onearg terms of Chapter 5 (see Sections 5.2.4 and 5.2.5).

The vararg operation takes a constructor τ and a function e. The typing rules constrain e to take a single argument of type τ. The operation calculates a coercion, based on τ, that turns e into a multi-argument function. In particular, if τ is a record constructor of the form Record_c[c1,...,cn], then the coercion is a function that takes n arguments of type c1,$\cdots$,cn, respectively. These arguments are placed into a record and passed to the original function e. If τ is not a record, then the coercion is the identity.

The onearg operation takes a constructor τ, a function e, and an argument e'. If τ is not a record constructor, then the operation simply applies e to e'. If τ is a record constructor of the form Record_c[c1,...,cn], then the e is constrained by the typing rules to be a multi-argument function that takes arguments of type c1,$\cdots$,cn, respectively, and e' is constrained to be a record of type τ. In this case, onearg extracts the components of the record e' and passes them directly as arguments to e.

The typing constraints on vararg and onearg are expressed using Typecase_c (see Section 5.2.1). For a fixed number of arguments, both vararg and onearg can be implemented directly in Lmli by a combination of term-level Typecase_e and Listcase_e expressions that deconstruct the argument constructor and calculate the appropriate coercion. Like peq, a later phase in the compiler replaces all occurrences of vararg and onearg with references to these terms, which are placed in a globally shared module. We leave both forms as primitive operations to facilitate optimization.

The Switch_e expression provides a combination of a control flow operator and a primitive deconstructor for integer, enum, sum, enum-or-record, enum-or-sum, and sum cursor values. The switch_exp argument to Switch_e is defined by the SML type abbreviation

```
type switch_exp =
        {switch_type : switch_type,
         arg         : exp,
         arms        : (word * function) list,
         default     : exp Option},
```

where switch_type is given by

```
datatype switch_type =
      Int_sw | Enum_sw | Sum_sw | Enumorrec_sw | Enumorsum_sw
    | Sumcursor_sw
```

Each clause or arm of the switch is indexed by a 32-bit word. For integer and enum switches, the value of the argument is used to select the appropriate arm according to this index. In these cases, the arms are functions that take no arguments. For sum switches, the enum in the first position of the sum is used to select the appropriate arm. The arm function must take a record type corresponding to the appropriate variant. For instance, if e has type Sum_c[[Int_c,String_c],[Int_c]], then the 0-arm must be a function that takes a record of type Record_c[Enum_c 2,Int_c,String_c] whereas the 1-arm must be a function that takes a record of type Record_c[Enum_c 2,Int_c]. If the switch is for an enum-or-record or enum-or-sum value, then the 0-arm corresponds to the enum case whereas the 1-arm corresponds to either a record or sum. In all cases, if an arm is missing, then the default expression is chosen. Defaults are required unless the arms are exhaustive.

Recall that sum cursor values are implemented as pairs consisting of an enum and a sum value. Switch expressions for sum cursor values are evaluated as follows: if the enum value of the cursor matches the enum value of the sum value, then the 0-arm is selected and the sum value is passed as an argument. If the enum value of the cursor does not match, then the 1-arm is selected and a new cursor value is constructed from the old. The new cursor has the same sum value, but increases the index by one. If the original sum cursor has type Sumcursor_c[c1,c2,...,cn], then the new sum cursor value has type Sumcursor_c[c2,...,cn]. Therefore, switch on sum cursors provides a means for eliminating one of the possible cases in a sum.

The `Tlistcase_e` and `Typecase_e` forms provide a simple eliminatory form for constructors at the term level. We do not provide `Fold_e` or `Typerec_e` at the term level, because these may be simulated with `Fixtype_d`. The argument to a `Tlistcase_e` is described by

```
type tlistcase_exp =
        {arg   : con,
         scheme: var * kind * con,
         nil_c : exp,
         cons_c: tfunction}.
```

The `arg` component is the argument constructor, which must be of a list kind. The `scheme` component is a type scheme used to describe the type of the clauses as well as the type of the entire expression. The type of the entire expression is obtained by substituting `arg` for the type variables of the scheme within the constructor of the scheme. The `nil_c` clause must have a type described by substituting `Nil_c` (at the appropriate kind) for the variable in the constructor. The `cons_c` clause must be a function that abstracts two constructor arguments, corresponding to the head and tail of the list of constructors.

The argument to a `Typecase_e` is described by

```
type typecase_exp =
        {arg     : con,
         scheme  : var * kind * con,
         arms    : (primcon * tfunction) list,
         mu_arm  : tfunction Option,
         default : tfunction Option}
```

Here, the `arg` component must be of kind `Mono_k`. Again, the entire type is obtained by substituting `arg` for the variable within the constructor of the scheme. Each arm is indexed by a primitive constructor. The `mu_arm` matches any `Mu_c` values. The "unrolling" of the recursive constructor is passed as an argument to this clause when it is selected. The `default` component is selected if the argument does not match any of the arms.

`Raise_e` raises an exception packet of type `Exn_c`, whereas `Handle_e(e,f)` evaluates `e` and if an exception is raised, the exception packet is passed to the function `f`. The function can use a combination of the `de_excon` primitive and a `Switch_e` to determine what exception was raised and extract a value from the packet.

Finally, the `Export_e` form is not an expression, but rather an anonymous module. Each compilation unit in TIL is constrained to be an expression consisting of a series of declarations that terminate with an `Export_e`. This form specifies a list of constructors and values that are to be exported by the module. Each module is given a globally unique `strid` M. The module is described by a signature

```
datatype signat =
    Signat of {types : (label * kind * (con Option)) list,
               values : (label * con) list}
```

that describes the kinds and types of the constructors and values exported by the module. Each exported constructor can optionally include the definition of the constructor in the signature. In this respect, Lmli signatures resemble the translucent sums of the Harper and Lillibridge module calculus [58]. The types of the values can contain references to the constructors exported by the module — using the `Dot_c` notation — relative to the module `strid` M. The module itself is represented with a datatype of the form

```
datatype module =
    Module of {name: strid,
               imports : (strid * signat) list,
               signat : signat,
               body : exp}.
```

The `imports` component specifies the set of modules (and their signatures) upon which this module depends.

8.5 Lambda to Lmli

In the translation from Lambda to Lmli, I eliminate datatype definitions and perform a series of type-directed transformations that specialize arrays and refs, flatten function arguments, box floating point values, and flatten certain representations of datatypes. All of these type-directed translations make use of dynamic type analysis when they encounter unknown types. I also provide Lmli terms that implement polymorphic equality, and the `vararg` and `onearg` primitives as suggested in Chapter 5.

In this section, I show how I compile datatypes to Lmli constructors, and discuss the various type-directed translations. I also contrast my approach to datatype representations with that of SML/NJ and show that, unlike SML/NJ, I am able to flatten data constructors without restricting abstraction.

8.5.1 Translating Datatypes

I translate a simple datatype definition of the form

datatype $(\alpha_1, \cdots, \alpha_n)$ `T` = D_1 | D_2 | $\cdots$ | D_m,

to a constructor function that abstracts the type variables (α_1 through α_n). The body of the function is a `Mu_c` constructor where `T` is bound to a representation of its definition (discussed below). For example, the SML datatype

$$\textbf{datatype}\ \alpha\ \texttt{tree = Leaf}\ \textbf{of}\ \alpha\ \texttt{|}\ \texttt{Node}\ \textbf{of}\ \alpha\ \texttt{tree * }\alpha\ \texttt{tree}$$

is compiled to a constructor function of the form

$$\texttt{tree = }\lambda\alpha\texttt{::Mono\_k.Mu\_c([(t,Sum\_c[[}\alpha\texttt{],[[Var\_c t,Var\_c t]]])],Var\_c t).}$$

(I have elided some kind information and used λ to represent the constructor function.) Within the definition of the datatype, I replace recursive references with the `Mu_c`-bound variable. For instance, in the previous tree definition, I replaced α `tree` with the variable `t`. This replacement is possible because I always verify that a datatype is applied to the same type variables that it abstracts (see Section 8.3). The resulting constructor has kind `Mono_k` $\rightarrow$ `Mono_k`.

The translation of a datatype applied to some type argument is straightforward: I simply apply the constructor function corresponding to the datatype to the translation of the type arguments.

This approach to datatypes was originally suggested by Harper [59], though he also suggests the use of an existential to hide the representation of the datatype. Hiding the representation of the datatype is important at the source level, since this distinguishes user types that happen to have the same representation. However, within the back-end of a compiler, there is no advantage to such abstraction. Therefore, I do not abstract the representations of datatypes.

The general form of an SML datatype definition is a series of mutually recursive definitions of the form

$$\begin{array}{lllll}
\textbf{datatype} & (\alpha_{1,1},\cdots,\alpha_{1,n_1}) & \texttt{T}_1 & = & D_{1,1}\ \texttt{|}\ D_{1,2}\ \texttt{|}\ \cdots\ \texttt{|}\ D_{1,m_1} \\
\textbf{and} & (\alpha_{2,1},\cdots,\alpha_{2,n_2}) & \texttt{T}_2 & = & D_{2,1}\ \texttt{|}\ D_{2,2}\ \texttt{|}\ \cdots\ \texttt{|}\ D_{2,m_2} \\
\quad\cdots & & & & \\
\textbf{and} & (\alpha_{p,1},\cdots,\alpha_{p,n_p}) & \texttt{T}_p & = & D_{p,1}\ \texttt{|}\ D_{p,2}\ \texttt{|}\ \cdots\ \texttt{|}\ D_{p,m_p}
\end{array}$$

In this general case, I generate one constructor function that abstracts all of the unique type arguments. The function uses a single `Mu_c` constructor to define the constructors simultaneously. I then export the set of constructors corresponding to the datatype as a constructor tuple. Individual types are obtained by instantiating the type variables and projecting the appropriate component from the tuple.

The representation that I choose for a datatype

$$\textbf{datatype}\ (\alpha_1,\cdots,\alpha_n)\ \texttt{T} = D_1\ \texttt{|}\ D_2\ \texttt{|}\ \cdots\ \texttt{|}\ D_m$$

depends on the form of the data constructors, D_1, D_2, $\cdots$, D_m. In SML, data constructors can have zero or one argument. I choose the representation of the datatype according to the following cases:

- If all of the data constructors take zero arguments, then we use an `Enum_c` as the translation of the datatype. For example, `datatype bool = true | false` is represented as an `(Enum_c 2)` constructor.

- If there is only one data constructor and this data constructor takes an argument of type τ, then we use the translation of τ as the representation of the datatype.

- If all of the data constructors take one argument, and there is more than one data constructor, we use a `Sum_c` (variant record) as the translation of the datatype. For example, `datatype foo = Bar of int | Baz of real` is translated to the constructor `Sum_c[[Int_c],[Real_c]]`.

- If all but one of the data constructors takes zero arguments, then we use an `Enumorrec_c` to represent the datatype. For example, the datatype α `list = nil | :: of` α `* (`α `list)` is translated to an `Enumorrec_c(1,[Record_c[`α`,Var_c t])` constructor (where `t` is recursively bound to the definition). The data constructor that takes an argument (e.g., `cons`) is always represented as a record so that it can be distinguished from `Enum_c` values. However, this introduces extra indirection when the argument to the data constructor is already a record (e.g., α `* (`α `list)`). A later phase eliminates this extra indirection when possible (see Section 8.5.3).

- If there is more than one data constructor that takes an argument and there are data constructors that take no arguments, the datatype is translated to an `Enumorsum_c` constructor. `Enum_c` values are used for the data constructors that take no arguments whereas `Sum_c` values are used for the data constructors that take arguments.

A naive representation of datatypes might map each datatype to a variant record. However, this approach would cause values such as `true`, `false`, and `nil` to be allocated. By mapping datatypes to the various efficient representation types, we avoid a great deal of allocation.

8.5.2 Specializing Arrays and Boxing Floats

During the translation from Lambda to Lmli, I translate polymorphic array operations such as `sub` and `update` to constructor abstractions that perform a `typecase` on the unknown type. The `typecase` selects the appropriate primitive operation (e.g., `psub Realarray`, `psub Intarray`, or `psub Ptrarray`) according to the instantiation of this type.

We chose to distinguish integer and floating point arrays from other kinds of arrays for a variety of reasons: first, in the presence of a generational collector, the update operation on these arrays does not require a write barrier, because the value placed in the array can never point across generational boundaries. Second, by leaving integer arrays untagged and unboxed, we are able to use them to represent both raw strings and byte arrays. Third, by distinguishing floating point arrays, we are able to align such arrays on 64-bit boundaries, providing efficient access to the elements of the array. SML/NJ provides specialized monomorphic strings, bytearrays, and floating point arrays for these same reasons. However, any array library function — such as an iterator — must be coded for each of these array types.

After translating Lambda to Lmli, I perform a series of type-directed translations on the resulting Lmli code. The first translation ensures that all float values are boxed (i.e., placed in a record), except when they are placed in floating point arrays. The translation simply boxes floating point literals, unboxes floats as they are passed to primitive operations (e.g., `pplus_r`), and boxes float results of primitive operations. Much of the boxing and unboxing is eliminated within a function by conventional optimization.

8.5.3 Flattening Datatypes

After boxing floats, I perform a type-directed translation to flatten `Enumorrec_c` values. Consider the list datatype:

datatype α `list = nil | ::` **of** α `* (`α `list)`

This datatype is initially translated to the Lmli constructor

```
list = λα::Mono_k.
          Mu_c([t,Enumorrec_c(1,[Record_c[α,Var_c t]])],Var_c t).
```

At this stage, list values will either be an `(Enum_c 1)` value corresponding to `nil` or a `Record_c[Record_c[`α`,list(`α`)]]` value corresponding to `cons`. Because the contents of a `cons` cell is always a record (`Record_c[`α`,list(`α`)]`), we can always determine such values from `nil` and thus eliminate the extra `Record_c[-]` in cons cells. After the constructor flattening phase, the list datatype is represented by the constructor

```
list = λα::Mono_k.Mu_c([t,Enumorrec_c(1,[α,Var_c t])],Var_c t).
```

This optimization eliminates an extra level of indirection in every cons cell, and is thus very important for typical SML code, which does a fair amount of list-processing.

However, we cannot always determine at compile time whether or not we can flatten an `Enumorrec_c` constructor. In particular, consider the option datatype:

datatype α `option = NONE | SOME` **of** α

The initial translation of this datatype yields the constructor

```
option = λα::Mono_k.Enumorrec_c(1,[α]).
```

Since the data constructor `SOME` has an argument of unknown type (α), we cannot determine whether `SOME` will always be applied to a record and thus cannot determine whether we can flatten the representation. In particular, the SML type `int option` should *not* be flattened because we cannot always tell integer values from `Enum_c` values.

Therefore, when we encounter an `Enumorrec_c(`n`,`α`)` constructor, we use `Typecase_c` on α to determine whether or not the constructor should be flattened. Therefore, after constructor flattening, the option datatype is represented by the constructor

```
option = λα::Mono_k.Typecase_c α of
                Record_c [τ1,···,τn] => Enumorrec_c(1,[τ1,···,τn])
              | _ => Enumorrec_c(1,[α])
```

When constructing an option value or deconstructing an option value, we must use `Typecase_e` at the term level to determine the proper code sequence.

My approach generalizes the constructor flattening performed in the SML/NJ compiler. In SML/NJ, cons cells are flattened but `SOME` cells are not, precisely because the compiler cannot determine at compile time whether it can safely flatten option datatypes. Even to support flattened cons cells, SML/NJ restricts the programmer from writing certain legal SML programs [10]. In particular, SML/NJ will not let the programmer abstract the contents of a cons cell in a signature as follows

```
signature LIST =
  sig
    type α Abstract_Cons
    datatype α list = nil | :: of α Abstract_Cons
  end

structure List : LIST =
  struct
    datatype α list = nil | :: of α Abstract_Cons
    withtype α Abstract_Cons = α * α list
  end.
```

Typing this code into the SML/NJ (version 1.08) interactive system yields the following message:

```
std_in:0.0-23.5 Error: The constructor :: of datatype list
has different representations in the signature and the structure.
Change the definition of the types carried by the constructors in
the functor formal parameter and the functor actual parameter so
that they are both abstract, or so that neither is abstract.
```

The problem is that a functor parameterized by the `LIST` signature cannot determine whether the contents of cons cells can be flattened; any such functor will be compiled assuming that cons cells are not flat, whereas the structure `List` will be compiled so that cons cells are flattened. My approach makes no such restriction because I dynamically determine the representation of abstract data structures when necessary.

8.5.4 Flattening Arguments

After flattening `Enumorrec_c` values, I flatten function arguments in the same manner as suggested in Chapter 5. If a function takes a record as an argument and the number of elements in the record does not exceed a constant k, then the function is transformed to take the elements of the record as multiple arguments. If a function takes an argument of known type that is not a record (or else the function takes a record with greater than k components[7]), then the function is not transformed. If a function takes an argument of unknown type, then the function is compiled expecting a single argument. The `vararg` primitive is used to calculate a coercion dynamically, based on the instantiation of the unknown type.

Likewise, an application is transformed so that, if the argument is a record and the number of elements in the record does not exceed k, then the components of the record are passed directly as multiple arguments. If the argument has known type, but is either not a record or else is a record of greater than k components, then the application is not transformed. If an application has an argument of unknown type, then I use the `onearg` primitive to calculate a coercion dynamically, based on the instantiation.

8.6 Bform and Optimization

After the translation to Lmli, and after the series of type-directed transformations, we translate Lmli to Bform. Bform is a subset of Lmli that makes an explicit distinction between small values and constructors that fit into registers, large values and constructors that must be allocated on the heap, and computations that produce values or constructors.

At the term level, small values are either variables, projections from a module, unit (an empty record), integers, floats, enums, external labels, or coercions applied to a small value. Large values include strings, records of small values, and functions. At the constructor level, small values are either variables, projections from modules, an empty tuple of constructors, or a 0-ary primitive constructor (e.g., `Int_c`). Large values include tuples and lists of constructors as well as primitive constructors that take arguments (e.g., `Record_c`).

[7] In the current prototype, k is arbitrarily set to eight arguments.

All large values and all computations are bound to variables in a declaration and we use the variable in place of the large value or computation. Thus, expressions always manipulate small values. These constraints ensure that we avoid duplicating large objects (such as strings) and preserve as much sharing as possible. These constraints also have the effect of linearizing nested computations and naming intermediate results. All of these constraints simplify standard optimization.

Numerous transformations and optimizations are applied in the Bform phase to programs. (See Tarditi [115] for a more complete description of these optimizations.) The optimizations include the following conventional transformations:

- **alpha-conversion:** We assign unique names to all bound variables.
- **minimizing fix:** We break functions into minimal sets of mutually recursive functions. This improves inlining, by separating non-recursive and recursive functions.
- **dead-code elimination:** We eliminate unreferenced, pure expressions, and functions.
- **uncurrying:** We transform curried functions to multi-argument functions whenever all of the call sites of the curried function can be determined.
- **constant folding:** We reduce arithmetic operations, switches, and typecases on constant values, as well as projections from known records.
- **sinking:** We push pure expressions used in only one branch of a switch into that branch.
- **inlining:** We always inline functions that are applied only once. We never inline recursive functions. We inline non-recursive, "small" functions in a bottom-up pass.
- **inlining switch continuations:** We inline the continuation of a switch, when all but one clause raises an exception. For example, the expression

  `let x = if y then` e_2 `else raise` e_3
  `in` e_4
  `end`

  is transformed to

  `if y then let x =` e_2 `in` e_4 `end else raise` e_3.

  This makes expressions in e_2 available within e_4 for optimizations such as common sub-expression elimination.

- **common subexpression elimination (CSE):** Given an expression

  ```
  let x = e1
  in e2
  end
  ```

  if e_1 is pure, then we replace all occurrences of e_1 in e_2 with `x`. Pure expressions include operations such as record projection that are guaranteed to terminate without effect, but exclude signed arithmetic (due to the possibility of overflow and divide-by-zero exceptions) and function calls.

- **eliminating redundant switches:** Given an expression

  ```
  let x = if z then
              let val y = if z then e1 else e2
              in ...
  ```

  we replace the nested `if` statement by e_1, since `z` is always `true` at that point.

- **hoisting invariant computations:** Using the call graph, we calculate the nesting depth of each function. We assign a `let`-bound variable and the expression it binds a nesting depth equal to that of the nearest enclosing function. For every pure expression e, if all free variables of e have a nesting depth less than e, we move the definition of e right after the definition of the free variable with the highest lexical nesting depth.

- **eliminating redundant comparisons:** We propagate a set of simple arithmetic relations of the form $x < y$ top-down through the program and a "rule-of-signs" abstract interpretation is used to determine signs of variables. We use this information to eliminate array-bounds checks and other tests.

In addition to these standard optimizations, I perform a set of type-specific optimizations:

- **eliminating peq:** If the polymorphic equality primitive is applied to a known type, and the number of syntax nodes in the type is smaller than some parameter, then I generate special equality code for that type. I delay performing this specialization until after hoisting and common subexpression elimination to avoid duplication.

- **eliminating vararg and onearg:** As suggested in Section 5.2.5, the `onearg` and `vararg` primitives cancel. I use this to eliminate applications of `onearg` and `vararg`. Also, as types become known, I specialize the `onearg` and `vararg` primitives to the appropriate coercion. As with `peq`, I delay performing this specialization until after hoisting and common subexpression elimination to avoid duplication.

- **hoisting type applications:** Because we make the restriction at the source level that all expressions assigned a $\forall$-type must be values (i.e., effect-free), we are assured that a type application is effect-free. Furthermore, the back-end does not introduce any polymorphic functions with computational effects, and thus all type applications are effect free. Therefore, like other pure operations, we hoist type applications.

Currently, we apply the optimization as follows. First, we perform a round of reduction optimizations, including dead-code elimination, constant folding, inlining functions called once, CSE, eliminating redundant switches, and invariant removal. These optimizations do not increase program size and should always result in better code. We iterate these optimizations until no further reductions occur. Then we perform switch-continuation inlining, sinking, uncurrying, comparison elimination, fix minimizing, and general inlining. The entire optimization process is then iterated for some adjustable number of times (currently three).

8.7 Closure Conversion

The closure conversion phase of TIL is based on the formal treatment of closure conversion given in Chapter 6, but following Kranz [78] and Appel [9], I extended the translation to avoid creating closures and environments unless functions "escape". A function escapes if it is placed in a data structure, passed as an argument to another function, or is returned as the result of a function. If a function does not escape, then all of its call sites can be determined and all of the free variables of the function are available at the call sites. Therefore, we transform non-escaping functions to code that takes all of their free variables as additional arguments, but we avoid creating an environment and closure for the function. Instead, we modify the call sites of each function to pass these extra values directly to the code.

A transformed call site may mention variables that occur free in the function being called, but not the original calling function. Therefore, we must take the set of applications to non-escaping functions into account when calculating the free variables of a function.

We use a flat constructor tuple to represent constructor environments, and a flat record to represent value environments. These environments are always allocated on the heap. To support recursion, we simultaneously define the environments and closures of a set of mutually recursive function using a Scheme-style "letrec" declaration.

TIL does not close over variables bound at the top level (i.e., outside of any function). Such variables are mapped to labels (machine addresses) by lower levels of the compiler and thus can be directly addressed. In practice, this results in a two-level environment, where a data pointer is used to access top-level values and a closure pointer is used to access values defined within a function. The advantage of this approach is that heap-allocated environments can be substantially smaller. The disadvantage is that values bound at the top-level cannot easily be garbage collected, since these values are bound to labels.

After closure conversion, we perform another round of optimization in an effort to clean up any inefficiencies introduced by closure conversion. Some optimizations, notably invariant removal and inlining, are turned off since they do not preserve the invariants of closure conversion.

8.8 Ubform, Rtl, and Alpha

After closure conversion and closure optimization, we translate the resulting code to Ubform. The Ubform intermediate language is an untyped Bform, but each variable is labelled with representation information. We erase the distinction between computations at the constructor and term levels in the translation to Ubform. Hence, constructor variables become term variables, constructor values become term values, and constructor computations become term computations. `Mono_k` constructors, such as `Int_c`, are represented as an enumeration, whereas `Mono_k` constructors that take an argument, such as `Record_c` are represented as tagged, variant records. Thus, the entire kind of `Mono_k` constructors is represented in the same fashion as an SML datatype.

The representation information on Ubform variables indicates whether each variable is an integer, float, pointer to a heap-allocated value, or of unknown representation at compile-time. `Enum_c`, `Enumorrec_c`, and `Enumorsum_c` values are considered to be pointers since the garbage collector can always determine whether they are in fact pointers at run time. Variables with unknown representation are annotated with *other* variables (corresponding to Bform type variables) that will contain the representation at run-time. An earlier stage boxes floating point values, thereby guaranteeing that variables of unknown representation are never floats. This invariant allows the register allocator to always assign a general purpose register to variables of unknown representation. Without the invariant, the register allocator would have to assign both a general purpose machine register and a floating point register to the variable and use dynamic type analysis to decide which of the two registers to use.

The Ubform representation is quite similar to a direct-style version of the CPS intermediate form used by Shao and Appel in the SML/NJ compiler [110]. While Shao and Appel claim that their compiler is type-based, they only use representation-based, untyped intermediate forms. Hence, it is not possible, in general, to verify automatically that any of their intermediate representations are type-safe. In contrast, only the last stages of TIL are untyped and a type checker can be used to verify automatically the type integrity of the code, even after optimization and closure conversion. Furthermore, the representation information we use at the Ubform level is more general than the representation information used by Shao and Appel, since we allow dynamic instantiation of representation information.

Currently, no optimization or other transformations occur at the Ubform level. We simply use the translation to Ubform as a convenient way to stage the compilation of the closure-converted code to the next intermediate form. This next form is called Rtl, which stands for Register Transfer Language. Rtl is similar to Alpha, MIPS, and other RISC-style assembly languages. However, it provides heavy-weight function call and return mechanisms, and a form of interprocedural goto for implementing exceptions. Rtl also provides an infinite number of pseudo-registers. In the conversion from Ubform to Rtl, we decide whether Ubform variables will be represented as constants, labels, or pseudo-registers. During the conversion, we also eliminate exceptions, insert tagging operations for records and arrays, and insert garbage collection checks.

The Rtl level would be suitable for a conventional, low-level imperative optimizer, similar to the ones found in C and Fortran compilers. We perform a few small optimizations, notably collapsing garbage collection checks and eliminating redundant loads of small constants.

Finally, the Rtl representation is translated to Alpha. Alpha is DEC Alpha assembly language, with extensions similar to those for Rtl. In the translation from Rtl to Alpha, we use conventional graph-coloring register allocation to allocate physical registers for the Rtl pseudo-registers. We also construct tables describing the layout and garbage collection information for each stack frame.

8.9 Garbage Collection

The translation from Ubform to Rtl and the translation from Rtl to Alpha, maintain the representation information that annotates variables. This representation information is used to construct tables for garbage collection. These tables tell the collector which registers and which stack slots contain pointers to heap-allocated objects.

Abstractly, we record enough information to determine which registers and which stack slots are live at every call site, and whether or not to trace these values, based on the representation information. We use the return address of call sites as an index to find the information and ensure that the return address is always saved in the first slot of a stack frame. In these respects, our collector closely resembles Britton's collector for Pascal [23] and the formal development of Chapter 7.

However, our collector is complicated by two details: the first complication is that some values have unknown representation at compile time. At the Ubform level, these values are labelled with another variable (corresponding to a type variable) that, at run time, indicates the representation of the value. Hence, for values of unknown representation, we must record where this other variable can be found so that the collector can determine whether or not to trace the original value. In this respect, our collector resembles Tolmach's tag-free collector for SML [119]. However, Tolmach calculates unknown representations lazily during garbage collection, because he does not have a general programming language at the type level. In particular, he only supports substitution at the constructor level and not, for instance, `Typerec`. In contrast, our constructor computations can perform type analysis, function call, and allocation. Therefore, we calculate unknown representations eagerly, during program evaluation so that all representations are already calculated before garbage collection is invoked.

The second complication is that we split the registers into a set of caller-saves and a set of callee-saves registers. Callee-saves registers are used to hold values needed after a procedure call. To use a callee-saves register as a temporary, a procedure must save the contents of the register on the stack and restore the contents before returning to the caller.

In effect, callee-saves registers are like extra arguments to a procedure that are simply returned with the result of the procedure. Unfortunately, the types and thus the representations of these extra arguments are unknown to the called procedure. We solve this issue by recording when callee-saves registers are saved into stack slots and when variables are placed into callee-saves registers. During garbage collection, we process the stack from oldest frame to youngest frame. Initially, the callee-saves registers are not live. If the first procedure places values into the callee-saves registers, then it knows the representations of these values. We propagate this information to the next stack frame. If the next procedure spills a callee-saves register to the stack, then we can determine the representation of the stack slot from the propagated representation information. Otherwise, we simply forward the representation information to the next stack frame, and so on. This approach to reconstructing type information is similar to the approach suggested by Appel [8] and Goldberg and Gloger [49, 50]. Once we determine which registers and which stack slots must be traced, we perform a standard copying garbage collection on the resulting roots. Currently, we use a simple two-generation collector.

8.10 Performance Analysis of TIL

In this section, I compare the performance of code produced by TIL against code produced by the SML/NJ compiler [12]. I also examine other aspects, including heap allocation, physical memory requirements, executable size, and compile time. The goal is to show that, for a reasonable set of benchmarks, TIL produces code that is comparable (or better) than the code produced by SML/NJ, at least for the subset of SML that TIL currently supports.

However, I make no attempt to compare TIL and SML/NJ except for these end-to-end measurements. There are many differences between these two systems, so we cannot directly compare particular implementation choices (such as whether or not to use tag-free collection), simply because we cannot fix all of the other variables. By showing that TIL code is comparable to SML/NJ, I hope to persuade the reader that a type-based implementation of SML that uses novel technologies, such as dynamic type dispatch and tag-free collection, can compete with one of the best existing ML compilers.

8.10.1 The Benchmarks

I chose a set of small to medium-sized benchmarks ranging from a few lines up to 2000 lines of code to measure the performance of TIL. Larger programs would be desirable, but there are few large SML programs that do not use nested modules or functors. Table 8.1 describes these programs. The benchmarks cover a range of application areas including scientific computing, list processing, systems programming, and compilers. Some of these programs have been used previously for measuring ML performance [9, 36]. Others were adapted from the Caml-Light distribution [24].

For this set of comparisons, I compiled all of the programs as single closed modules. For `lexgen` and `simple`, which are standard benchmarks [9], I eliminated functors by hand, since TIL does not yet support functors.

For TIL, I compiled programs with all optimizations enabled. For SML/NJ, I compiled programs using the default optimization settings. I used a recent internal release of SML/NJ (a variant of version 108.3), since it produces code that is about 35% faster than the standard 0.93 release of SML/NJ [110].

For both compilers, we extended the built-in types with safe 2-dimensional arrays. The 2-d array operations perform bounds checking on each dimension and then use unsafe 1-d array operations. Arrays are stored in column-major order.

TIL automatically prefixes a set of operations onto each module that it compiles. This "inline" prelude is about 280 lines in length. It contains 2-d array operations, commonly-used list functions, and so forth. By prefixing the module with these definitions, we ensure that they are exposed to the optimizer. To avoid handicapping SML/NJ, I created separate copies of the benchmark programs for SML/NJ, and carefully placed equivalent "prelude" code at the beginning of each program.

Since TIL creates stand-alone executables, I used the `exportFn` facility of SML/NJ to create stand-alone programs. The `exportFn` function of SML/NJ dumps part of the heap to disk and throws away the interactive system.

8.10.2 Comparison against SML/NJ

I compared the performance of TIL against SML/NJ in several dimensions: execution time, total heap allocation, physical memory footprint, the size of the executable, and

Program	lines	Description
cksum	241	Checksum fragment from the Foxnet [18], doing 5000 checksums on a 4096-byte array, using a stream interface [17].
dict	166	Insert 10,000 strings, indexed by integers into a balanced binary tree, lookup each string and replace it with another. The balanced binary trees are taken from the SML/NJ library.
fft	246	Fast-Fourier transform.
fmult	63	Matrix multiply of two 100x100 floating point matrices.
imult	63	Matrix multiply of two 200x200 integer matrices.
kb	618	The Knuth-Bendix completion algorithm.
lexgen	1123	A lexical-analyzer generator [13], processing the lexical description of SML/NJ.
life	146	A simulation of cells implemented using lists [103].
logic	459	A simple Prolog-like interpreter, with unification and backtracking.
msort	45	List merge sort of 5,120 integers, 40 times.
pia	2065	A Perspective Inversion Algorithm [125] deciding the location of an object in a perspective video image.
qsort	141	Integer array quicksort of 50,000 pseudo-random integers, 2 times.
sieve	27	Sieve of Eratosthenes, filtering primes up to 30000.
simple	870	A spherical fluid-dynamics program [39], run for 4 iterations with grid size of 100.
soli	131	A solver for a peg-board game.

Table 8.1: Benchmark Programs

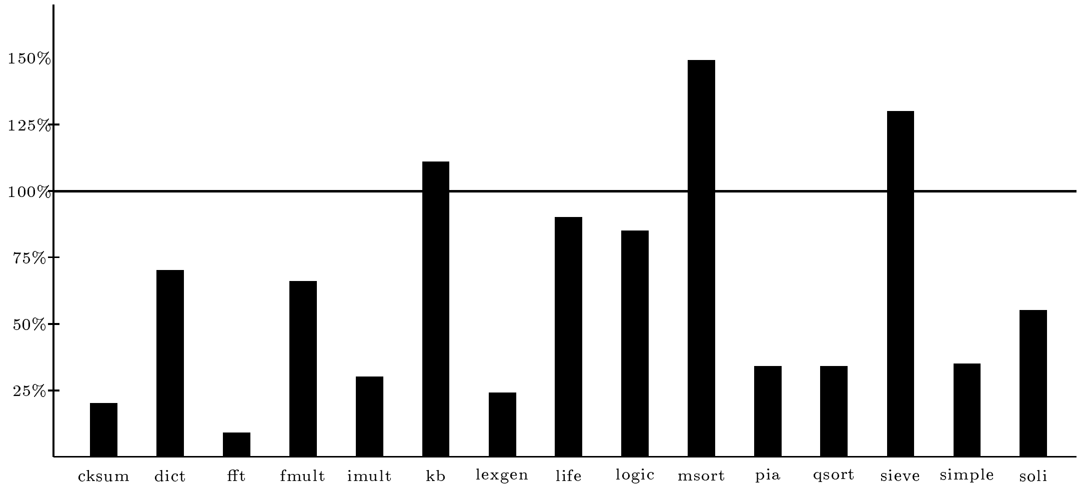


Figure 8.4: TIL Execution Time Relative to SML/NJ

compilation time.

I measured execution time on a DEC Alpha AXP 250-4/266 workstation, running OSF/1, version V3.2, using the UNIX `getrusage` function. For SML/NJ, I started timing after the heap had been reloaded. For TIL, I measured the entire execution time of the process, including load-time. I made 5 runs of each program on an unloaded workstation and chose the lowest execution time. The workstation had 96 Mbytes of physical memory, so paging was not a factor in the measurements.

I measured total heap allocation by instrumenting the TIL runtime to count the bytes allocated. I used existing instrumentation in the SML/NJ run-time system. I measured the maximum amount of physical memory during execution using `getrusage`.

To compare program sizes, I first compiled empty programs under TIL and under SML/NJ. The empty program for TIL generates a stripped executable that is around 250 Kbytes, whereas the empty program for SML/NJ consists of roughly 425 Kbytes from the runtime, and 170 Kbytes from the heap, for a total of 595 Kbytes. Next, I stripped all executables produced by TIL, and then subtracted the size of the empty program (250 Kbytes) from the size of each program. For SML/NJ, I measured the size of the heap generated by `exportFn` for each program and subtracted the size of the heap generated by the empty program (170 Kbytes).

Finally, I measured end-to-end compilation time, including time to assemble files produced by TIL and time to export a heap image for SML/NJ.

Figures 8.4 through 8.8 present the measurements. The raw numbers appear in Tables 8.2 through 8.6. For each benchmark, measurements for TIL were normalized to those for SML/NJ and then graphed. SML/NJ represents the 100% mark on all the graphs,

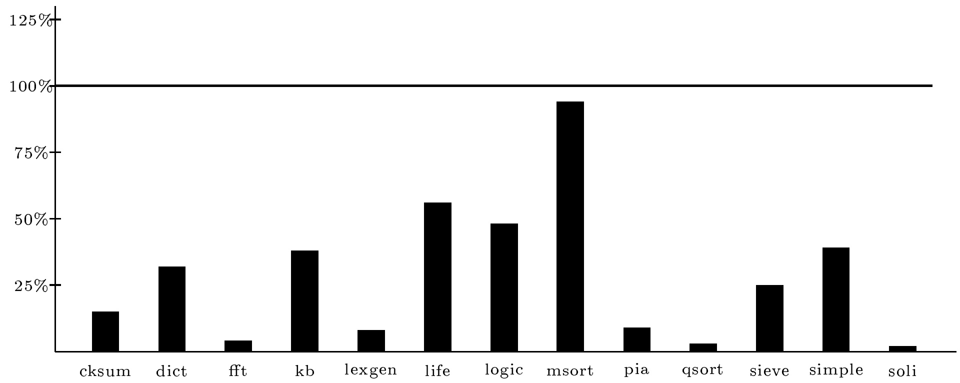


Figure 8.5: TIL Heap Allocation Relative to SML/NJ (excluding `fmult` and `imult`)

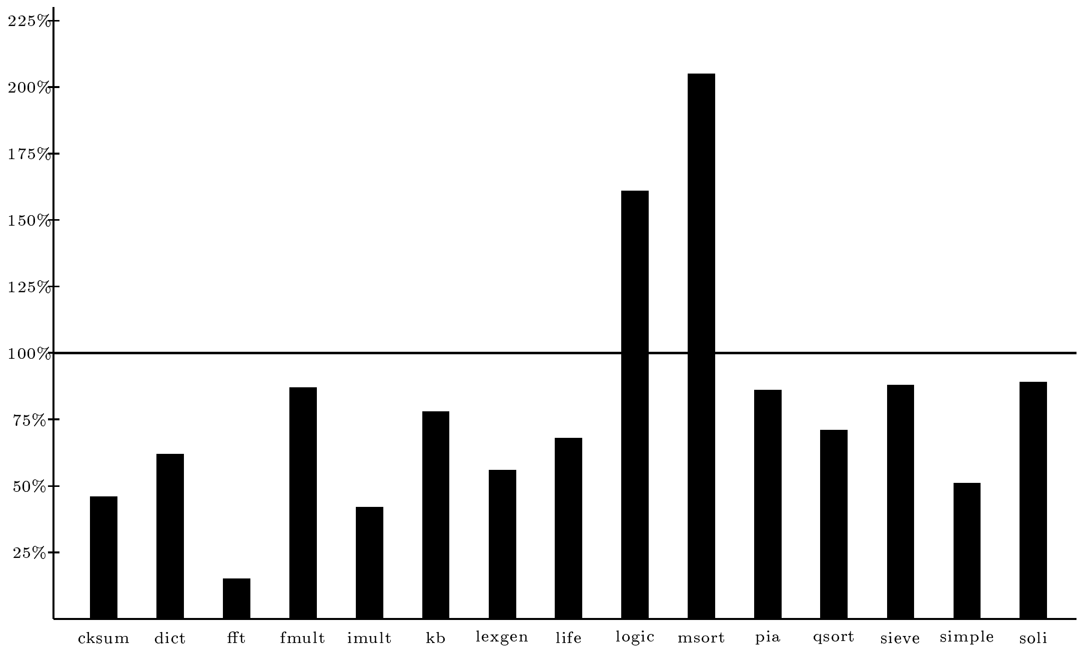


Figure 8.6: TIL Physical Memory Used Relative to SML/NJ

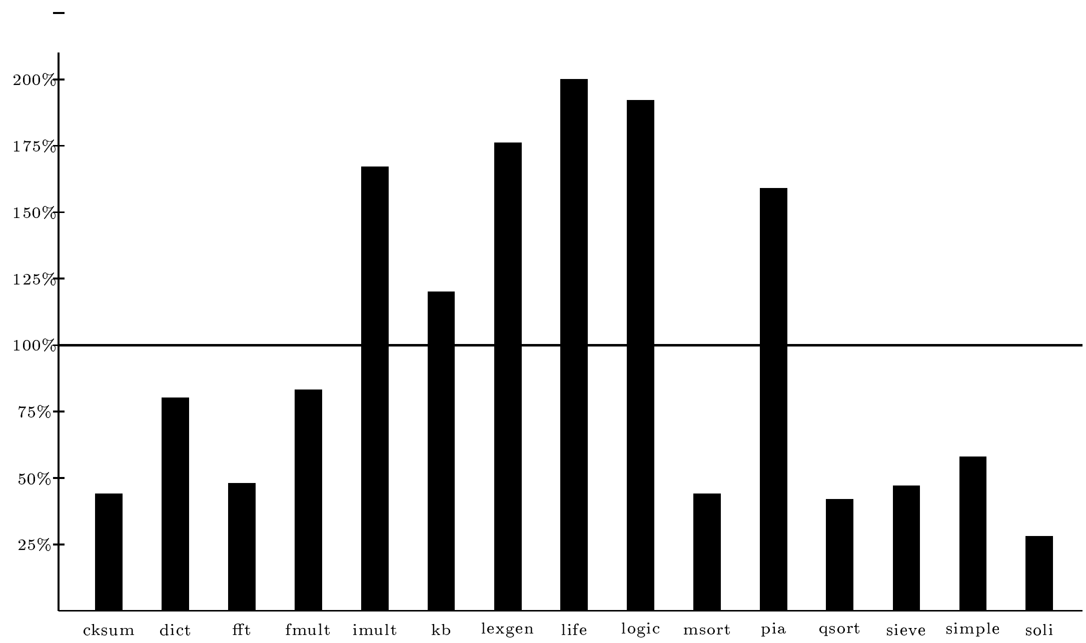


Figure 8.7: TIL Executable Size Relative to SML/NJ (without runtimes)

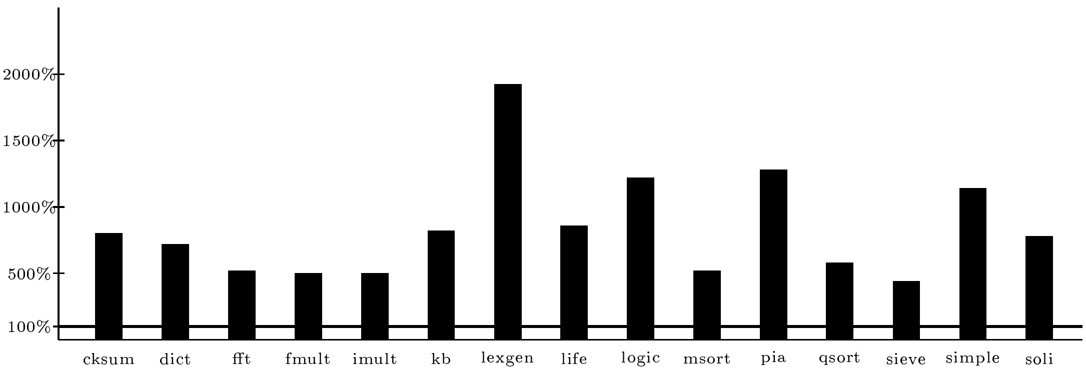


Figure 8.8: Til Compilation Time Relative to SML/NJ

indicated by a solid horizontal line.

Figure 8.4 presents relative running times. On average, programs compiled by TIL run about two times faster than programs compiled by SML/NJ. All programs but `kb` and `msort` run faster under TIL than under SML/NJ. Furthermore, the TIL versions of the largest programs, `lexgen`, `pia`, and `simple`, are more than twice as fast as their SML/NJ counterparts. Finally, the slowest program, `msort`, is no more than 50% slower than the SML/NJ version.

The `kb` benchmark uses exceptions and exception handlers quite frequently. TIL does a relatively poor job of register saving and restoring around exception handlers and I suspect that this is the reason for its poor performance on this benchmark. The poor performance of `msort` is most likely due to the difference in garbage collectors for the two systems, since roughly two-thirds of the running time for TIL is spent in the collector. I speculate that the multi-generational collector of SML/NJ does a better job of memory management for this benchmark.

Since SML/NJ flattens arguments using Leroy-style coercions, and also flattens some constructors (see Section 8.5.3), the primary difference in performance for most benchmarks is *not* due to my type-directed translations. Most likely, the primary difference in performance is due to the conventional optimizations that TIL employs [115]. What is remarkable is that, even though TIL employs more optimizations than SML/NJ, the use of types and dynamic type dispatch does not interfere with optimization. Furthermore, for some benchmarks (notably `fft` and `simple`) much, if not all, of the performance improvement is due to the type-directed array flattening (see Section 8.10.4). Regardless, a reasonable conclusion to draw from these measurements is that type-directed compilation and dynamic type dispatch does not interfere with optimization and, when coupled with a good optimizer, yields code that competes quite well with existing compilers.

Figure 8.5 compares the relative amounts of heap allocation between TIL and SML/NJ, except for the `fmult` and `imult` benchmarks. The TIL version of `fmult` allocates over 16 Mbytes of data, whereas the SML/NJ version allocates less around 1 Kbyte. This is entirely because TIL does not flatten floating point values into registers across function calls. During the dot product loop of the TIL version, floating point values are pulled out of the arrays, multiplied, and added to an accumulator, and the accumulator is boxed as it is passed around the loop. Under SML/NJ, the accumulator remains unboxed. In contrast, the TIL version of `imult` does not allocate at all at run time, whereas the SML/NJ version allocates about 1 Kbyte. Even including `fmult` but excluding `imult`, the geometric mean of the ratios of heap-allocated data shows that TIL programs allocate about 34% of the amount of data that SML/NJ allocates. This low percentage is not surprising, because TIL uses a stack for activation records, whereas SML/NJ allocates activation records on the heap.

Figure 8.6 presents the relative maximum amounts of physical memory used. TIL programs tend to use either much less or much more memory than SML/NJ programs. I speculate that some variability is due to the different strategies used to size the heaps. SML/NJ uses a multi-generational collector with a heap-size-to-live-data ratio of 3 to 1 for older generations, whereas TIL uses a two-generation collector that has a heap-size-to-live-data ratio of up to 10 to 1 (the ratio varies). Also, since TIL does not yet properly implement tail-recursion (tail calls within exception handlers are not implemented properly) the stack may be larger than it needs to be.

Figure 8.7 compares executable sizes, excluding runtimes and any pervasives. On average, TIL programs are about 80% of the size of SML/NJ programs, and no program is more than twice as big as the SML/NJ version. These sizes confirm that generating tables for nearly tag-free garbage collection consumes a modest amount of space. The numbers also establish that the inlining strategy used by TIL produces code of reasonable size.

Figure 8.8 compares compilation times for TIL and SML/NJ. SML/NJ does quite a lot better than TIL when it comes to compilation time, compiling about six times faster. However, we have yet to tune TIL for compilation speed. Most of the compile time is spent in the optimizer and the register allocator. We assume that much of the time in the register allocator can be eliminated by using an intelligent form of coalescing as suggested by George and Appel [45]. We assume that much of the time spent in the optimizer can be eliminated by simply tuning and inlining key routines.

Another reason the optimizer is slow is that we always fully normalize a type whenever we want to determine some property (e.g., the domain or range of an arrow type). Normalizing is an expensive process that destroys a great deal of sharing. By lazily normalizing, we hope to improve many optimization phases that depend upon type information.

Finally, I speculate that a great deal of time and allocation during compilation is due to our naive approach of maintaining type information. In particular, we label each bound value variable with its type, and we label each bound type variable with its kind. In the B-form representation, this means that almost every construct has associated type information and this type information contains a great deal of kind information. Much of the type/kind information is unneeded or can easily be recovered. Many primitive transformations, such as α-conversion, must process this unneeded information and are thus slowed by the inefficient representation.

8.10.3 The Effect of Separate Compilation

In this section, I explore the effect that separate compilation has on the performance of some of the benchmarks. When programs are separately compiled, the optimizer cannot perform as many reductions and transformations. Hence, the likelihood that the resulting program will use dynamic dispatch increases when compared to the same program compiled all together.

I took two of the larger programs, `logic` and `lexgen`, and broke them into modules at natural boundaries, resulting in the benchmark programs `logic_s` and `lexgen_s`. These benchmarks should give an indication of how TIL performs with realistic, separately compiled programs.

I also took the `dict`, `fmult`, `imult`, and `msort` benchmarks, placed the core routines into separate modules, and abstracted the types and primitive operations of the routines. Thus, these modules provide a set of generic library routines and abstract datatypes (balanced binary trees, matrix multiply, list sort) that can be used at any type. During development, programmers are likely to use such modules. I wanted to determine what the costs are of holding the types abstract and separately compiling the modules from their uses.

Table 8.7 describes the resulting benchmarks. The running times of these benchmarks relative to SML/NJ are graphed in Figure 8.9, and the raw numbers are given in Table 8.8. On the whole, the TIL programs run roughly as well or better than their SML/NJ counterparts. Only `logic_s`, `msort0` and `msort1` are slower, and by no more than 20%.

Figure 8.10 compares the running times of each of the separately compiled programs to the comparable, globally compiled benchmark of the previous section. For the non-matrix benchmarks, we see about a 10-20% overhead in separate compilation. For the matrix benchmarks, we see over a 350% overhead. The difference between the `fmult0`/`imult0` and `fmult1`/`imult1` bars is because `fmult0` and `imult0` must perform dynamic type dispatch to select an array operation, whereas `fmult1` and `imult1` do not. Hence, we see that most of the overhead of separate compilation is not due to type abstraction, but rather to the fact that the primitive operations (multiplication and addition) are held abstract.

These figures indicate that TIL provides a tradeoff between separate compilation and performance. During development, programmers can use separate compilation and expect that their code will perform reasonably well. Towards the end of development, as key routines are identified through profiling, programmers can specialize the types of generic routines and expect a modest gain in performance, without sacrificing full separate compilation. Clients of a specialized generic abstraction need only be re-compiled if the type exported by that abstraction changes. At the very end of development, when the most important abstractions and routines are identified, programmers can inline these modules to get the best performance, but at the cost of separate compilation.

8.10.4 The Effect of Flattening

In this section, I explore the performance effect of the various type-directed flattening translations in TIL. Of course, we cannot easily determine the entire impact of types on the system. For instance, it is impossible to determine what effect the tag-free garbage

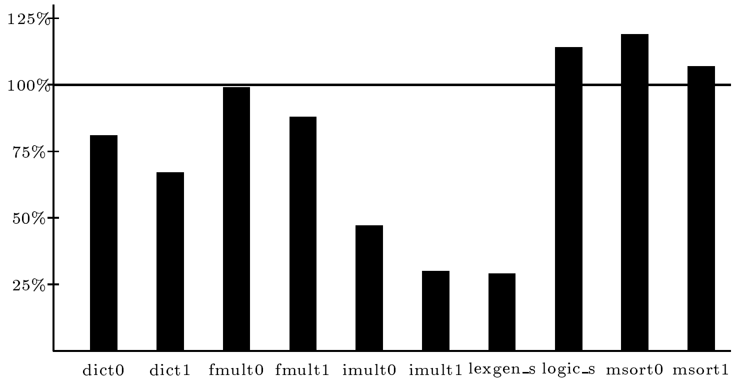


Figure 8.9: TIL Execution Time Relative to SML/NJ for Separately Compiled Programs

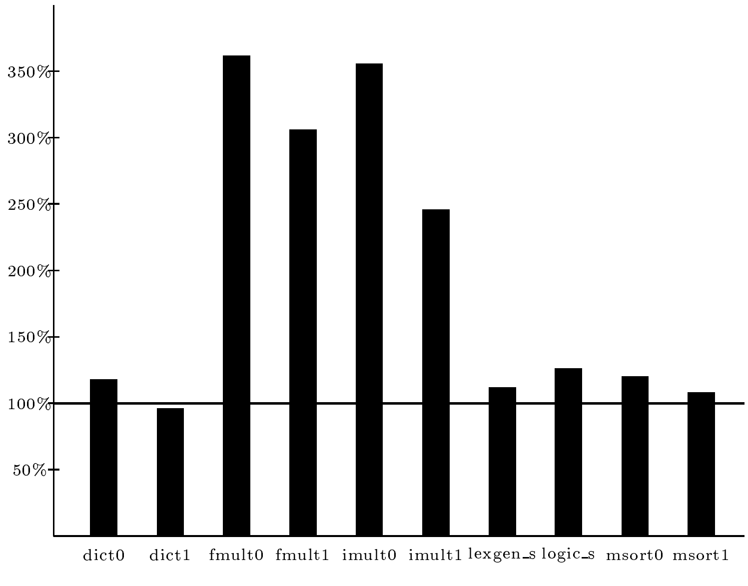


Figure 8.10: Execution Time of Separately Compiled Programs Relative to Globally Compiled Programs

collector has without building a corresponding tagging collector. Therefore, I have only examined those uses of types that I can easily turn off and on.

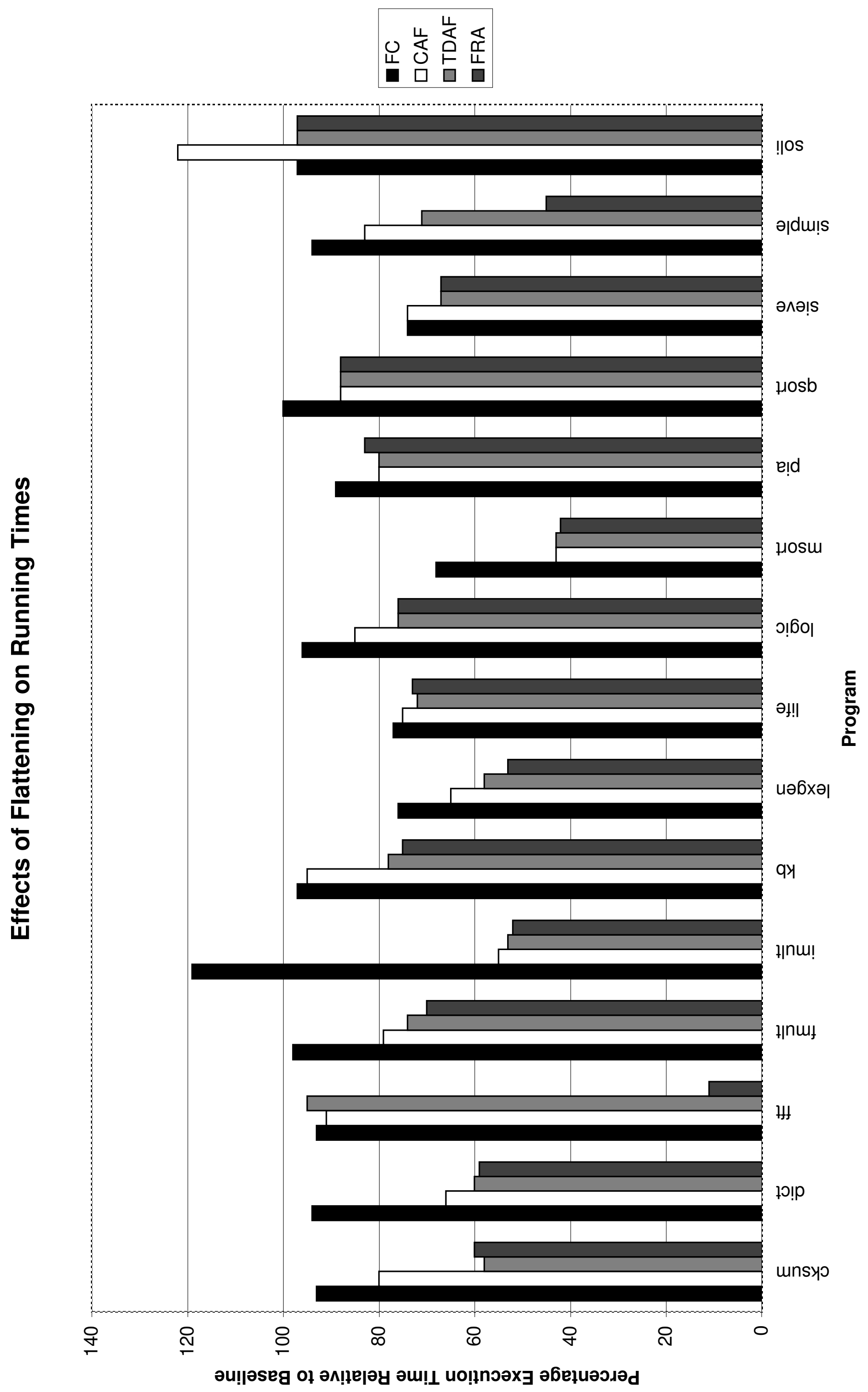
For each of the benchmarks described in the previous section, I measured the running time and amount of heap allocation of the program when compiled in the following ways:

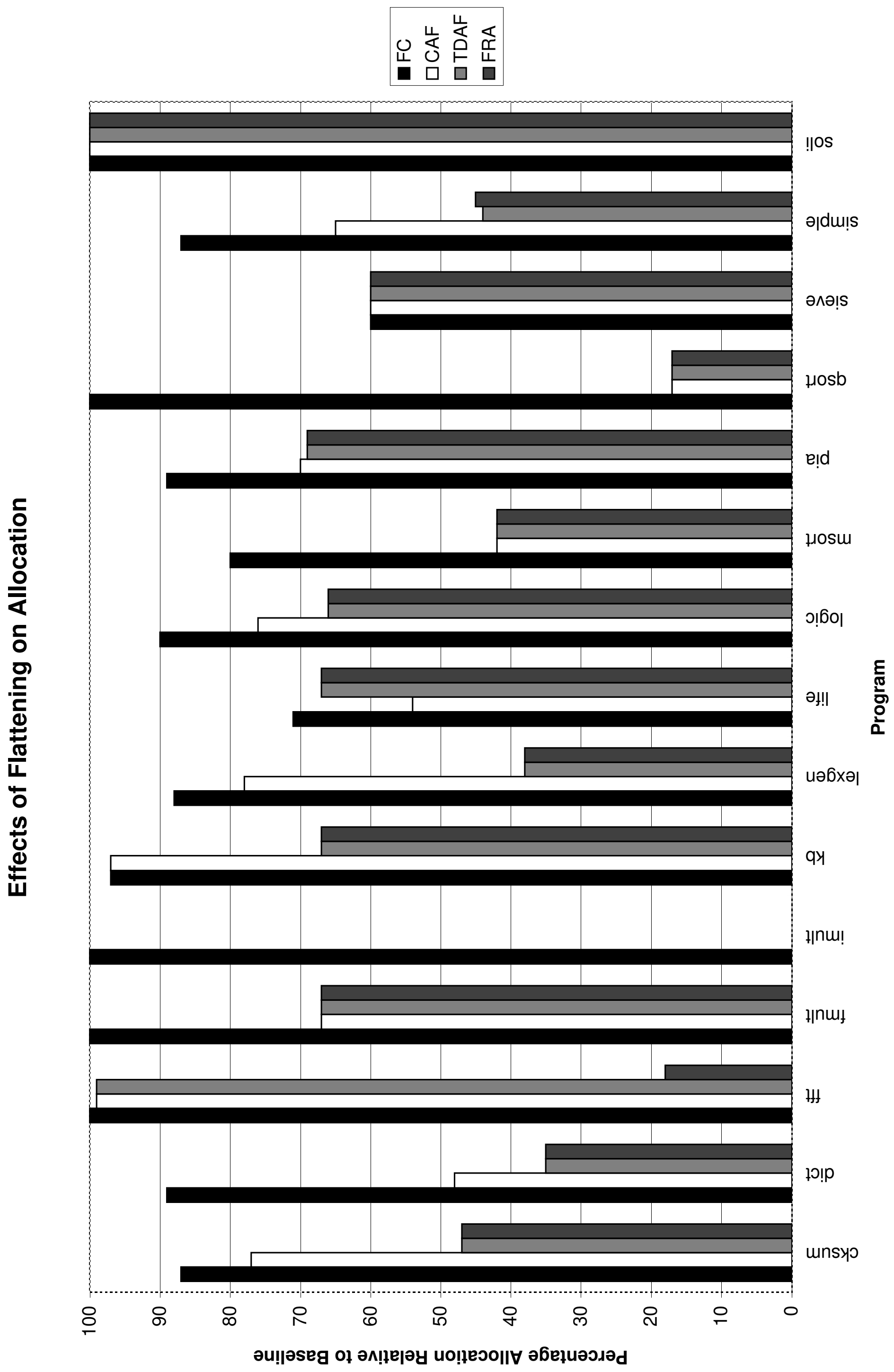
- **B** (Baseline): We do not flatten arguments, constructors, or arrays.
- **FC** (Flattened Constructors) : We flatten all `Enumorrec_c` constructors that contain records. This effectively flattens lists and option datatypes. Dynamic type dispatch is used when the component type is unknown.
- **CAF** (Conventional Argument Flattening): In addition to **FC**, We examine the call sites of each non-escaping function. If each call-site applies the function to a known record, then we flatten the function and pass the components of the record directly as arguments.
- **TDAF** (Type-Directed Argument Flattening): In addition to **FC**, we flatten all functions that take records as arguments, and flatten all applications of functions to records. We use dynamic type dispatch when the argument type is unknown.
- **FRA** (Flattened Real-Arrays): In addition to **TDAF**, we flatten all polymorphic arrays of floating point values. We use dynamic type dispatch for array operations, when the component type is unknown.

Tables 8.9 and 8.10 record the running times (in seconds) and amounts of heap allocation (in megabytes) for each program compiled in each configuration. The numbers in parenthesis indicate the ratio to the corresponding baseline. Figure 8.10.4 plots the running times, normalized to the baseline; Figure 8.10.4 plots the allocation, normalized to the baseline.

From the data, we can conclude that flattening both constructors and arguments is almost always worthwhile, both in terms of running times and allocation. All together, the flattening phases provide an average speedup of 42% and decrease allocation by 50%. The biggest improvements for most benchmarks comes from argument flattening. Furthermore, type-directed argument flattening does as well if not better than conventional argument flattening in almost all cases, providing an addition speedup of 7% and an additional decrease in allocation of 9%, on average. This is in part because type-directed flattening is able to flatten higher-order functions, whereas conventional argument flattening cannot.

The increase in running time for `imult`, when constructors are flattened, appears to be an anomaly in the measurements. Separate and longer runs (10 times each) indicate that constructor flattening has no measurable effect at all on running times or allocation, but the original data shows a 19% increase in running times.


Effects of Flattening on Running Times
Percentage Execution Time Relative to Baseline
0
20
40
60
80
100
120
140
Program
cksum
dict
fft
fmult
imult
kb
lexgen
life
logic
msort
pia
qsort
sieve
simple
soli
FC
CAF
TDAF
FRA


Effects of Flattening on Allocation
Percentage Allocation Relative to Baseline
Program
FC
CAF
TDAF
FRA
cksum
dict
fft
fmult
imult
kb
lexgen
life
logic
msort
pia
qsort
sieve
simple
soli
0
10
20
30
40
50
60
70
80
90
100

Flattening real arrays has mixed results on most benchmarks, causing allocation or running times to vary up or down slightly. This is not surprising since most of the benchmarks do not manipulate floating point arrays; when floating point arrays are flattened, these benchmarks must perform dynamic type dispatch when working with other array types. However, some benchmarks that do manipulate floating point arrays, notably `fft` and `simple`, show dramatic speedups: `fft` shows an 84% improvement in running time and an 88% reduction in allocation. Surprisingly, the amount of allocation for `simple` increases, but running times decrease. I speculate that, since we box floating point values passed to other functions, this accounts for the increased allocation, since values pulled out of a flattened array must be boxed before being passed to a function. A similar effect happens for `fmult`. Furthermore, these boxes are short-lived — lasting only a function call — and are thus not preserved by the garbage collector. In contrast, when values are boxed before being placed in an array, the boxes may tend to live longer. Also, as boxed arrays are updated, the generational collector must be informed of any potential generational conflicts. This may account for the fact that `simple` allocates more, but runs faster when floating point arrays are flattened. Regardless, since flattening real arrays has a negligible negative effect on the other benchmarks, it is a worthwhile optimization.

All of these results are consistent with results seen by Shao and Appel [110]. The advantages of my approach are that (a) we can flatten data constructors without making restrictions at the source level, (b) we can flatten arrays, (c) we need not tag values for garbage collection or polymorphic equality.

Program	Exec. time (s)		TIL/NJ
	TIL	NJ	
`cksum`	2.36	11.68	0.20
`dict`	0.40	0.57	0.70
`fft`	1.39	15.67	0.09
`fmult`	0.33	0.50	0.66
`imult`	1.46	4.93	0.30
`kb`	1.93	1.74	1.11
`lexgen`	0.65	2.76	0.24
`life`	1.29	1.44	0.90
`logic`	7.98	9.42	0.85
`msort`	2.72	1.82	1.49
`pia`	0.38	1.11	0.34
`qsort`	0.44	1.31	0.34
`sieve`	0.39	0.30	1.30
`simple`	8.51	24.02	0.35
`soli`	0.31	0.56	0.55
Geo. mean			0.50

Table 8.2: Comparison of TIL Running Times to SML/NJ

Program	Heap alloc. (Kbytes)		TIL/NJ
	TIL	NJ	
`cksum`	143.897	984.775	0.15
`dict`	12.445	38.495	0.32
`fft`	9.108	214.853	0.04
`fmult`	16.000	0.001	16,000.00
`imult`	0.000	0.001	0.00
`kb`	36.941	96.761	0.38
`lexgen`	8.753	113.405	0.08
`life`	25.447	45.259	0.56
`logic`	253.053	525.997	0.48
`msort`	114.052	121.738	0.94
`pia`	5.238	55.142	0.09
`qsort`	1.035	35.332	0.03
`sieve`	2.525	7.282	0.35
`simple`	323.394	826.504	0.39
`soli`	0.328	15.606	0.02
Geo. mean (excluding `imult`)			0.39

Table 8.3: Comparison of TIL Heap Allocation to SML/NJ

Program	Phys. mem. (Kbytes)		TIL/NJ
	TIL	NJ	
`cksum`	672	1472	0.46
`dict`	1152	1872	0.62
`fft`	2672	17592	0.15
`fmult`	816	936	0.87
`imult`	504	1208	0.42
`kb`	2712	3480	0.78
`lexgen`	1672	2992	0.56
`life`	816	1208	0.68
`logic`	6576	4096	1.61
`msort`	10032	4896	2.05
`pia`	1376	1592	0.86
`qsort`	1096	1536	0.71
`sieve`	2256	2576	0.88
`simple`	9088	17784	0.51
`soli`	1000	1120	0.89
Geo. mean			0.69

Table 8.4: Comparison of TIL Maximum Physical Memory Used to SML/NJ

Program	Exec. size (Kbytes)		TIL/NJ
	TIL	NJ	
`cksum`	32.768	73.840	0.44
`dict`	24.576	30.720	0.80
`fft`	40.960	85.128	0.48
`fmult`	163.840	196.632	0.83
`imult`	327.680	196.632	1.67
`kb`	90.112	74.880	1.20
`lexgen`	271.336	153.824	1.76
`life`	40.960	20.480	2.00
`logic`	98.304	51.272	1.92
`msort`	8.192	18.432	0.44
`pia`	237.568	149.728	1.59
`qsort`	16.384	38.936	0.42
`sieve`	8.192	17.408	0.47
`simple`	188.416	325.808	0.58
`soli`	16.384	58.424	0.28
Geo. mean			0.81

Table 8.5: Comparison of TIL Stand-Alone Executable Sizes to SML/NJ (excluding runtimes)

Program	Comp. time (s)		TIL/NJ
	TIL	NJ	
`cksum`	12.89	1.62	7.96
`dict`	11.44	1.57	7.29
`fft`	11.24	2.13	5.28
`fmult`	3.88	0.77	5.04
`imult`	3.88	0.78	4.97
`kb`	59.42	7.19	8.26
`lexgen`	262.52	13.60	19.3
`life`	21.15	2.48	8.53
`logic`	85.40	7.01	12.18
`msort`	3.79	0.73	5.19
`pia`	205.16	15.93	12.89
`qsort`	7.27	1.25	5.82
`sieve`	2.52	0.57	4.42
`simple`	206.46	18.27	11.30
`soli`	10.52	1.35	7.79
Geo. mean			5.8

Table 8.6: Comparison of TIL Compilation Times to SML/NJ

Program	Description
`dict0`	Generic dictionary structure, with key and value types held abstract as well as key comparison function. Instantiated with integer keys and string values as in the `dict` benchmark.
`dict1`	Same as `dict0`, but with types known. Only the key comparison is held abstract.
`fmult0`	Generic matrix multiply routine, with element type held abstract as well as primitive multiplication, addition, and zero values. Instantiated with floating point type and values as in the `fmult` benchmark.
`fmult0`	Same as `fmult0`, but with the element type known (real). Only the primitive multiplication, addition, and zero values are held abstract.
`imult0`	Generic matrix multiply routine, with element type held abstract as well as primitive multiplication, addition, and zero values. Instantiated with integer type and values as in the `imult` benchmark.
`imult1`	Same as `imult0`, but with the element type known (int). Only the primitive multiplication, addition, and zero values are held abstract.
`lexgen_s`	Same as `lexgen` benchmark, but broken into separately compiled modules.
`logic_s`	Same as `logic` benchmark, but broken into separately compiled modules.
`msort0`	Generic list merge sort, with element type held abstract as well as comparison operator. Instantiated with integer type and comparison as in the `msort` benchmark.
`msort1`	Same as `msort0`, but with the element type known (int). Only the comparison operator is held abstract.

Table 8.7: Separately Compiled Benchmark Programs

Program	Comp. time (s)		TIL/NJ
	TIL	NJ	
dict0	0.47	0.58	0.81
dict1	0.38	0.57	0.67
fmult0	1.19	1.20	0.99
fmult1	1.01	1.15	0.88
imult0	5.19	7.57	0.67
imult1	3.58	7.57	0.47
lexgen_s	0.72	2.49	0.29
logic_s	9.99	8.74	1.14
msort0	3.24	2.73	1.19
msort1	2.93	2.73	1.07

Table 8.8: Comparison of TIL Execution Times Relative to SML/NJ for Separately Compiled Programs

Program	Execution time in seconds (ratio to baseline)				
	B	**FC**	**CAF**	**TDAF**	**FRA**
cksum	3.92	3.65 (0.93)	3.15 (0.80)	2.29 (0.58)	2.36 (0.60)
dict	0.68	0.64 (0.94)	0.45 (0.66)	0.41 (0.60)	0.40 (0.59)
fft	12.26	11.41 (0.93)	11.20 (0.91)	11.60 (0.95)	1.39 (0.11)
fmult	0.47	0.46 (0.98)	0.37 (0.79)	0.35 (0.74)	0.33 (0.70)
imult	2.80	3.32 (1.19)	1.54 (0.55)	1.47 (0.53)	1.46 (0.52)
kb	2.58	2.49 (0.97)	2.45 (0.95)	2.01 (0.78)	1.93 (0.75)
lexgen	1.23	0.94 (0.76)	0.80 (0.65)	0.71 (0.58)	0.65 (0.53)
life	1.77	1.37 (0.77)	1.33 (0.75)	1.28 (0.72)	1.29 (0.73)
logic	10.45	10.07 (0.96)	8.87 (0.85)	7.92 (0.76)	7.98 (0.76)
msort	6.45	4.37 (0.68)	2.80 (0.43)	2.75 (0.43)	2.72 (0.42)
pia	0.46	0.41 (0.89)	0.37 (0.80)	0.37 (0.80)	0.38 (0.83)
qsort	0.50	0.50 (1.00)	0.44 (0.88)	0.44 (0.88)	0.44 (0.88)
sieve	0.58	0.43 (0.74)	0.43 (0.74)	0.39 (0.67)	0.39 (0.67)
simple	18.99	17.94 (0.94)	15.75 (0.83)	13.40 (0.71)	8.52 (0.45)
soli	0.32	0.31 (0.97)	0.39 (1.22)	0.31 (0.97)	0.31 (0.97)
Geom. mean		(0.90)	(0.77)	(0.70)	(0.58)

Table 8.9: Effects of Flattening on Running Times

Program	Allocation in Mbytes (ratio to baseline)				
	B	**FC**	**CAF**	**TDAF**	**FRA**
`cksum`	307.99	267.02 (0.87)	205.35 (0.77)	143.90 (0.47)	143.90 (0.47)
`dict`	35.66	31.73 (0.89)	17.02 (0.48)	12.45 (0.35)	12.45 (0.35)
`fft`	51.48	51.48 (1.00)	51.00 (0.99)	51.00 (0.99)	9.11 (0.18)
`fmult`	24.00	24.00 (1.00)	16.00 (0.67)	16.00 (0.67)	16.00 (0.67)
`imult`	96.00	96.00 (1.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
`kb`	54.96	53.52 (0.97)	53.24 (0.97)	36.94 (0.67)	36.94 (0.67)
`lexgen`	22.77	20.13 (0.88)	17.87 (0.78)	8.75 (0.38)	8.75 (0.38)
`life`	37.80	26.98 (0.71)	20.51 (0.54)	25.45 (0.67)	25.45 (0.67)
`logic`	384.37	345.49 (0.90)	292.07 (0.76)	253.05 (0.66)	253.05 (0.66)
`msort`	270.93	215.70 (0.80)	114.05 (0.42)	114.05 (0.42)	114.05 (0.42)
`pia`	7.64	6.80 (0.89)	5.33 (0.70)	5.24 (0.69)	5.24 (0.69)
`qsort`	6.06	6.06 (1.00)	1.04 (0.17)	1.04 (0.17)	1.04 (0.17)
`sieve`	4.21	2.53 (0.60)	2.53 (0.60)	2.53 (0.60)	2.53 (0.60)
`simple`	717.65	627.65 (0.87)	469.02 (0.65)	316.41 (0.44)	323.39 (0.45)
`soli`	0.33	0.33 (1.00)	0.33 (1.00)	0.33 (1.00)	0.33 (1.00)
Geom. mean (excluding `imult`)		(0.88)	(0.65)	(0.56)	(0.50)

Table 8.10: Effects of Flattening on Allocation

Chapter 9

Summary, Future Work, and Conclusions

In this thesis, I have demonstrated that compiler writers can take advantage of types for everything from enhancing performance to proving correctness. The fundamental idea behind my approach is to use a combination of *type-directed translation* and *dynamic type dispatch* to build a language implementation. Type-directed translation provides a formal framework for specifying and proving the correctness of compiler transformations, whereas dynamic type dispatch provides a means for applying type-directed translation to languages with unknown or variable types.

9.1 Summary of Contributions

I have presented a core calculus, called λ_i^{ML}, that provides dynamic type dispatch at both the term and the constructor levels. I have shown that type-checking λ_i^{ML} is decidable and that the type system is sound with respect to the operational semantics.

I gave examples of type-directed translations for SML-like languages to λ_i^{ML}-like languages. These translations demonstrated how function arguments and data structures could be flattened, how tag-free ad-hoc operations such as polymorphic equality could be implemented, how the constraints of Haskell-style type classes could be encoded, and how communication primitives could be strongly typed, yet dynamically instantiated. I also demonstrated how to prove correctness of these translations using logical relations.

I showed how a key transformation in functional language implementation, *closure conversion*, could be implemented as a type-directed and type-preserving translation, even for languages like λ_i^{ML}. I also proved the correctness of this translation.

I developed a formal, yet intuitive framework for expressing program evaluation that makes the heap, stack, and registers explicit. This model of evaluation allowed me to address memory management issues that higher-level models leave implicit. I gave a general definition of garbage and a general specification of trace-based garbage collectors. I proved that such collectors do not interfere with evaluation. I then showed how types could be used to derive shape information during garbage collection, thereby obviating the need to place tags on values at run time. I proved the correctness of this tag-free collection algorithm for monomorphic languages and showed how to extend the technique to λ_i^{ML}-like languages. My formulation was at a sufficiently abstract level that the proofs were tractable, yet the formulation was not so abstract that important details were lost.

Together with others, I constructed a compiler for SML called TIL to explore the practical issues of type-directed translation and dynamic type dispatch. This compiler uses typed intermediate languages based on λ_i^{ML} for almost all optimizations and transformations. TIL uses type-directed translation and dynamic type dispatch to flatten arguments, to generate efficient representations of datatypes, and to specialize arrays. TIL also uses dynamic type dispatch to support partially tag-free garbage collection and tag-free polymorphic equality. Finally, for a wide range of programs, the code emitted by TIL is as good or better than code produced by Standard ML of New Jersey.

9.2 Future Work

Because this thesis explores so many aspects of types and language implementation, from proving compiler correctness to implementing tag-free garbage collection, there are many unresolved issues among each of the topics. In this section, I discuss those issues that I feel are the most important.

9.2.1 Theory

From both a type-theoretic standpoint, one of the most interesting open issue for λ_i^{ML} is extending the language to support dynamic type dispatch on recursive types at the constructor level. This would allow us to reify a much wider class of type translations as constructor terms. For instance, the datatype flattening used in TIL can only be expressed at the meta-level (i.e., in the TIL compiler) and not as a constructor within the intermediate language Lmli. However, a straightforward extension of Typerec to generally recursive types is difficult, as this is likely to break constructor normalization, and hence decidability of type checking.

Of a related nature, the restriction to predicative polymorphism is suitable for interpreting ML-like languages. However, this restriction prevents us from compiling languages based on the original Girard-Reynolds impredicative calculus. Girard has shown that adding Typerec-like operators to such calculi breaks strong normalization [47], so it is unlikely that there is a simple calculus that provides both decidable type checking and impredicative polymorphism. In an impredicative calculus, recursive types (μ), $\forall$-types, and $\exists$-types can all be viewed as primitive constructors of kind $(\Omega \to \Omega) \to \Omega$, so to some degree, the ability to analyze recursive and polymorphic types requires some intensional elimination form for functions.

Whereas λ_i^{ML} provides a convenient intermediate form within a compiler, its use as a source language is somewhat problematic. The issue is that dynamic type analysis makes no distinction between user-level abstract types that happen to have the same representation. From a compiler perspective, the whole purpose of dynamic type dispatch is to violate the very abstraction that a programmer establishes. There are a variety of approaches that could be taken to solve this problem. The work of Duggan and Ophel [38] and Thatte [116] on kind-based definitions of type-classes seems promising to me. The basic idea is to allow users to define new inductively generated kinds that could be refinements or extensions of the kind of monotypes and, using a combination of type dispatch and methods corresponding to the new constructors, allow users to define appropriate elim forms at both the constructor and term levels.

In our previous work on closure conversion [92], we showed how closures could be represented using a combination of translucent types and existentials. Using this representation in TIL would allow us to hoist code and environment projections out of loops, but would greatly complicate the type system. In particular, the target language of closure conversion would need to be impredicative, and as mentioned earlier, this is problematic when combined with dynamic type analysis. However, I suspect that there is a simpler formulation that offers the same performance benefits without requiring full translucent sums and existentials.

Finally, all of these extensions make the underlying proof theory much more difficult. For instance, in an impredicative setting, we must use some technique like Girard's method of candidates — instead of simple, set-based logical relations — to prove translation correctness. Providing simple formulations of these techniques is imperative.

9.2.2 Practice

From a practical standpoint, we now know that we can generate good code for polymorphic languages if types are readily available at compile time. Furthermore, we know that for at least the applications studied here, types are either known for the most part, or can be made to be known. However, I expect that the degree of polymorphism in programs will only increase as more programmers start to use advanced languages. Hence, the next logical step is to explore techniques to make polymorphism fast without constraining the performance of monomorphic code. For example, it would be very profitable in terms of execution time to hoist `typecase` expressions out of loops, in an effort to get good code within the loops. However, hoisting `typecase` out of a loop requires that we duplicate the body of the loop for each arm of the `typecase`. This may be entirely reasonable for small loops, such as the dot product of matrix multiply, where we have a small number of cases in the `typecase`. However, in general, I feel that some sort of profile-driven feedback mechanism is needed to determine which `typecase`s should be hoisted.

Another practical point that needs to be addressed is the issue of unboxing floating point values as function arguments and within data structures other than arrays. Unboxing floating point values in function arguments using the `vararg`/`onearg` approach is problematic when there are a large number of argument registers and the underlying machine has split integer and floating point registers. The problem is that for k arguments, `vararg` must be able to generate 2^k coercions to deal with all of the possible calling conventions. This approach is impractical if k is greater than a fairly small constant (i.e., 5 or 6).

Fortunately, there is an alternative approach. When `vararg` is applied to a function f that is expecting one argument, we generate a closure that contains a runtime routine that works as follows: when the routine is called, it spills all of the possible argument registers to the stack. Then, using the type of f, the routine determines which registers actually contained arguments. Next, the routine allocates a record on the heap and copies the argument values into the record. Finally, the routine calls f passing this record to the function. A primitive corresponding to `onearg` would have the opposite functionality. These primitives are likely to be more expensive than the tailored conversions that TIL currently uses, and for conventional SML code — which rarely manipulates floating point values — it is not clear that the overheads would justify the costs.

There are enough tradeoffs in data representations that it is not clear that, for instance, flattening floating point values in records would be worth the cost. Certainly, flattening floating point values in arrays has mixed benefits. There are other issues that should be addressed as well, including alignment, bit fields, "endian-ness", word size, and so forth. Fortunately, TIL provides an excellent framework to explore these representation tradeoffs.

Clearly, the compile times of TIL are a current problem, and we have yet to address this issue. Initial tests confirm that the size of the type information on intermediate terms is quite large. However, most optimizations, including common sub-expression elimination, do *not* optimize types that decorate terms. Rather, the optimizer only processes constructors that are bound via a `let`-construct at the term level. By extending the optimizations to process types, it may be possible to reduce the size significantly, and hence the compile times of terms. Alternatively, we could use a representation where as little type and kind information as is possible remains on terms, and reconstruct this information as needed.

There are a wealth of issues to explore with respect to tag-free garbage collection. For example, it would be good to implement a fully tagging and fully tag-free implementation of TIL to explore the costs and benefits of our current approach. Fortunately, we abstracted many details of the garbage collection implementation in the higher levels of the compiler so that these experiments would someday be possible.

9.3 Conclusions

For compiler writers, types provide a means of encapsulating complex invariants and analysis information. Type-directed translation shows how we can take this analysis information from the source-level and transform it, with the program, so that intermediate levels can take advantage of this information. In this respect, the type system of λ_i^{ML} is far more powerful than conventional polymorphic calculi because it encapsulates *control-flow* information (i.e., Typerec). However, unlike a fully reflective language, λ_i^{ML} is sufficiently restricted that we can automatically normalize types and compare them. These restrictions make proofs of compiler correctness tractable, and tools like the type checker for TIL's intermediate forms possible.

A key advantage that type-directed translation has over traditional compiler transformations is that, for languages like SML, type information is readily available. Programmers must specify the types of imported values and often these types do not involve variables. I took advantage of this property in TIL to perform argument, constructor, and array flattening. These type-based transformations are in no way inhibited by higher-order functions or modules. In contrast, transformations based on data-flow, control-flow, or set-based analyses often fail to optimize terms due to a lack of information. For example, the conventional argument flattener of TIL fails to flatten many functions that the type-directed flattener does flatten. Even without programmer-supplied type information, the advances in soft typing [64, 7, 29, 132] provide a means for compiler writers to take advantage of types.

In general, compilers and other kinds of system software have real issues and problems that can serve as the clients and driving force behind the development of advanced type systems. Changing an intermediate language in a compiler to take advantage of recent advances is much more tractable than changing a ubiquitous source language. As type systems become more advanced, more information will be available to compilers, enabling more aggressive transformations. Thus, the real future for both type theory and compilers is in their marriage.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J.Lévy. Explicit substitutions. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, Jan. 1990.

[2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J.Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, Oct. 1991.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 213–227, San Francisco, Jan. 1990.

[4] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Progamming Languages and Systems*, 13(2):237–268, Apr. 1991. Revised version of [3].

[5] S. Aditya and A. Caro. Compiler-directed type reconstruction for polymorphic languages. In *ACM Conference on Functional Programming and Computer Architecture*, pages 74–82, Copenhagen, June 1993.

[6] S. Aditya, C. Flood, and J. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *ACM Conference on Lisp and Functional Programming*, pages 12–23, Orlando, June 1994.

[7] A. Aiken, E. L. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Jan. 1994.

[8] A. W. Appel. Runtime tags aren't necessary. *LISP and Symbolic Computation*, 2:153–162, 1989.

[9] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[10] A. W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–429, Oct. 1993.

[11] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Jan. 1989.

[12] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science.*

[13] A. W. Appel, J. S. Mattson, and D. Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[14] H. Baker. Unify and conquer (garbage, updating, aliasing ...) in functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 218–226, Nice, 1990.

[15] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1984.

[16] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science 1993, Bombay*, New York, 1993. Springer-Verlag. Extended abstract.

[17] E. Biagioni. Sequence types for functional languages. Technical Report CMU-CS-95-180, School of Computer Science, Carnegie Mellon University, Aug. 1995. Also published as Fox Memorandum CMU-CS-FOX-95-06.

[18] E. Biagioni, R. Harper, P. Lee, and B. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Conference on Lisp and Functional Programming*, pages 55–64, Orlando, June 1994.

[19] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit, Version 1. Technical Report 93/14, Department of Computer Science (DIKU), University of Copenhagen, 1993.

[20] G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Apr. 1993.

[21] H.-J. Boehm. Space-efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, June 1993.

[22] P. Branquart and J. Lewi. A scheme for storage allocation and garbage collection for Algol-68. In *Algol-68 Implementation.* North-Holland Publishing Company, Amsterdam, 1970.

[23] D. E. Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[24] Caml light. `http://pauillac.inria.fr:80/caml/`.

[25] L. Cardelli. Phase distinctions in type theory. Unpublished manuscript.

[26] L. Cardelli. The functional abstract machine. *Polymorphism*, 1(1), 1983.

[27] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan. 1995.

[28] L. Cardelli. A language with distributed scope. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 286–297, San Francisco, Jan. 1995.

[29] R. Cartwright and M. Fagan. Soft typing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–292, Toronto, June 1991.

[30] J. Chirimar, C. A. Gunter, and J. G. Riecke. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, San Francisco, June 1992.

[31] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *ACM Conference on Lisp and Functional Programming*, pages 13–27, 1986.

[32] R. L. Constable, *et. al. Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, 1986.

[33] C. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In *ACM Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64, Nancy, Sept. 1985. Springer-Verlag.

[34] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, Jan. 1990.

[35] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, June 1992.

[36] A. Diwan, D. Tarditi, and E. Moss. Memory-system performance of programs with intensive heap allocation. *Transactions on Computer Systems*, Aug. 1995.

[37] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, Jan. 1995.

[38] D. Duggan and J. Ophel. Kinded parametric overloading. Technical Report CS-94-35, University of Waterloo, Department of Computer Science, September 1994. Supersedes CS-94-15 and CS-94-16, March 1994, and CS-93-32, August 1993.

[39] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[40] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *Third Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Aug. 1986.

[41] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[42] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, June 1993.

[43] P. Fradet. Collecting more garbage. In *ACM Conference on Functional Programming and Computer Architecture*, pages 24–33, Orlando, June 1994.

[44] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, June 1991. ACM.

[45] L. George and A. W. Appel. Iterated register coalescing. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, Jan. 1996. To appear.

[46] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application a l'elimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium, edited by J.E. Fenstad. North-Holland, Amsterdam*, pages 63–92, 1971.

[47] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur.* PhD thesis, Université Paris VII, 1972.

[48] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[49] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–176, Toronto, June 1991.

[50] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In *ACM Conference on Lisp and Functional Programming*, pages 53–65, San Francisco, June 1992.

[51] J. Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, Jan. 1995.

[52] C. A. Gunter, E. L. Gunter, and D. B. MacQueen. Computing ML equality kinds using abstract interpretation. *Information and Computation*, 107(2):303–323, Dec. 1993.

[53] C. Hall, K. Hammond, S. Peyton-Jones, and P. Wadler. Type classes in Haskell. In *Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256. Springer-Verlag, 1994.

[54] J. Hannan. A type system for closure conversion. In *The Workshop on Types for Program Analysis*, Aarhus University, May 1995.

[55] R. Harper. Strong normalization and confluence for predicative, higher-order intensional polymorphism. Unpublished note.

[56] R. Harper and P. Lee. Advanced languages for systems software: The Fox project in 1994. Technical Report CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, Jan. 1994.

[57] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, Jan. 1993.

[58] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Jan. 1994.

[59] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Progamming Languages and Systems*, 15(2):211–252, April 1993. (See also [93].).

[60] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, Jan. 1990.

[61] R. Harper and G. Morrisett. Compiling with non-parametric polymorphism (preliminary report). Technical Report CMU-CS-94-122, School of Computer Science, Carnegie Mellon University, Mar. 1994. Also published as Fox Memorandum CMU-CS-FOX-94-03.

[62] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[63] R. W. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation*, 6:361–379, 1993.

[64] F. Henglein. Global tagging optimization by type inference. In *ACM Conference on Lisp and Functional Programming*, pages 205–215, San Francisco, June 1992.

[65] F. Henglein and J. Jørgensen. Formally optimal boxing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 213–226, Portland, Jan. 1994. ACM.

[66] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Progamming Languages and Systems*, 4(4):527–551, Oct. 1982.

[67] P. Hudak. A semantic model of reference counting and its abstraction. In *ACM Conference on Lisp and Functional Programming*, pages 351–363, Aug. 1986.

[68] P. Hudak, S. L. P. Jones, and P. Wadler. Report on the programming language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[69] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *ACM Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, Sept. 1985. Springer-Verlag.

[70] M. B. Jones, R. F. Rashid, and M. R. Thompson. Matchmaker: An interface specification language for distributed processing. In *Twelfth ACM Symposium on Principles of Programming Languages*, pages 225–235, New Orleans, Jan. 1985.

[71] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Currently available as Technical Monograph PRG-106, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, U.K. email: library@comlab.ox.ac.uk.

[72] M. P. Jones. A theory of qualified types. In *ESOP '92: European Symposium on Programming, Rennes, France*, New York, February 1992. Springer-Verlag. Lecture Notes in Computer Science, 582.

[73] M. P. Jones. Partial evaluation for dictionary-free overloading. Research Report YALEU/DCS/RR-959, Yale University, New Haven, April 1993.

[74] M. P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, May 1994.

[75] S. P. Jones and J. Launchbury. Unboxed values as first-class citizens. In *ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Cambridge, Sept. 1991. ACM, Springer-Verlag.

[76] R. Kelsey and P. Hudak. Realistic compilation by program translation – detailed summary –. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Jan. 1989.

[77] F. Knabe. *Language Support for Mobile Agents.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[78] D. Kranz et al. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 1986.

[79] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[80] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1966.

[81] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, Jan. 1992.

[82] X. Leroy. Polymorphism by name. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 220–231, Charleston, Jan. 1993.

[83] X. Leroy. Manifest types, modules, and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Jan. 1994.

[84] B. Liskov. Overview of the Argus language and system. Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, Feb. 1984.

[85] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, Mar. 1988.

[86] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 111–122, Austin, Nov. 1987. ACM.

[87] P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics, pages 81–109. North-Holland, 1975.

[88] D. Matthews. Poly manual. *ACM SIGPLAN Notices*, 20(9):42–76, 1985.

[89] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[91] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. Technical Report CMU-CS-95-171, School of Computer Science, Carnegie Mellon University, July 1995. Also published as Fox Memorandum CMU-CS-FOX-95-05.

[92] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*. ACM, Jan. 1996. To appear.

[93] J. Mitchell and R. Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, pages 28–46, San Diego, Jan. 1988.

[94] J. C. Mitchell and R. Harper. The essence of ML. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 28–46, Jan. 1988.

[95] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. Technical Report CMU-CS-95-110, School of Computer Science, Carnegie Mellon University, Jan. 1994. Also published as Fox Memorandum CMU-CS-FOX-95-01.

[96] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[97] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Progamming Languages and Systems*, 13(3):342–371, July 1991.

[98] S. Nettles. A Larch specification of copying garbage collection. Technical Report CMU–CS–92–219, School of Computer Science, Carnegie Mellon University, Dec. 1992.

[99] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 154–165, Albuquerque, Jan. 1992.

[100] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 99–112, Charleston, Jan. 1993.

[101] A. Ohori and T. Takamizawa. A polymorphic unboxed calculus as an abstract machine for polymorphic languages. Technical Report RIMS-1032, RIMS, Kyoto University, May 1995.

[102] E. R. Poulsen. Representation analysis for efficient implementation of polymorphism. Technical report, Department of Computer Science (DIKU), University of Copenhagen, Apr. 1993. Master Dissertation.

[103] C. Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, 1989.

[104] J. Reynolds. Types, abstraction, and parametric polymorphism. In *Proceedings of Information Processing '83*, pages 513–523, 1983.

[105] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the Annual ACM Conference*, pages 717–740, 1972.

[106] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation. Lecture Notes in Computer Science*, volume 19, pages 408–425. Springer-Verlag, Berlin, 1974.

[107] M. Serrano and P. Weis. 1+1 = 1: an optimizing Caml compiler. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 101–111, Orlando, June 1994. INRIA RR 2265.

[108] Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, 1994.

[109] Z. Shao and A. W. Appel. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming*, pages 150–161, Orlando, June 1994.

[110] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995.

[111] G. L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.

[112] P. Steenkiste and J. Hennessey. Tags and type checking in LISP: Hardware and software approaches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 50–59, Oct. 1987.

[113] S. Stenlund. *Combinators, λ-terms and Proof Theory.* D. Reidel, 1972.

[114] B. Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley, 1991.

[115] D. R. Tarditi. *Optimizing ML.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[116] S. R. Thatte. Semantics of type classes revisited. In *ACM Conference on Lisp and Functional Programming*, pages 208–219, Orlando, June 1994.

[117] M.-F. Thibault. *Représentations des Fonctions Récursives Dans les Catégories.* PhD thesis, McGill University, Montreal, 1977.

[118] P. J. Thiemann. Unboxed values and polymorphic typing revisited. In *ACM Conference on Functional Programming and Computer Architecture*, pages 24–35, La Jolla, 1995.

[119] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, June 1994.

[120] D. Ungar. Generational scavenging: A non-disruptive high performance storage management reclamation algorithm. In *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 15–167, Pittsburgh, Apr. 1984.

[121] United States Department of Defense. *Reference Manual for the Ada Programming Language*, Feb. 1983. U.S. Government Printing Office, ANSI/MIL-STD-1815A-1983.

[122] P. Wadler and S. Blott. How to make ad hoc polymorphism less ad hoc. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, 1989.

[123] L. R. Walmer and M. R. Thompson. A programmer's guide to the Mach user environment. School of Computer Science, Carnegie Mellon University, Feb. 1988.

[124] M. Wand and P. Steckler. Selective and lightweight closure conversion. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 435–445, Portland, Jan. 1994.

[125] K. G. Waugh, P. McAndrew, and G. Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, Aug. 1990.

[126] P. Weis, M.-V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

[127] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, Sept. 1992. Springer-Verlag.

[128] P. R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [127]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps.` In revision, to appear.

[129] P. Wodon. Methods of garbage collection for Algol-68. In *Algol-68 Implementation.* North-Holland Publishing Company, Amsterdam, 1970.

[130] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, Apr. 1991.

[131] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Department of Computer Science, Rice University, Feb. 1993.

[132] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. In *ACM Conference on Lisp and Functional Programming*, pages 250–262, Orlando, June 1994.