PROGRAMMING WITH TYPES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Stephanie Claudene Weirich

August 2002

PROGRAMMING WITH TYPES

Stephanie Claudene Weirich, Ph.D.
Cornell University 2002

Run-time type analysis is an increasingly important linguistic mechanism in modern programming languages. Language runtime systems use it to implement services such as accurate garbage collection, serialization, cloning and structural equality. Component frameworks rely on it to provide reflection mechanisms so they may discover and interact with program interfaces dynamically. Run-time type analysis is also crucial for large, distributed systems that must be dynamically extended, because it allows those systems to check program invariants when new code and new forms of data are added. Finally, many generic user-level algorithms for iteration, pattern matching, and unification can be defined through type analysis mechanisms.

However, existing frameworks for run-time type analysis were designed for simple type systems. They do not scale well to the sophisticated type systems of modern and next-generation programming languages that include complex constructs such as first-class abstract types, recursive types, objects, and type parameterization. In addition, facilities to support type analysis often require complicated language semantics that allow little freedom in their implementation. This dissertation investigates the foundations of run-time type analysis in the context of statically-typed, polymorphic programming languages. Its goal is to show how such a language may support type-analyzing operations in a way that balances expressiveness, safety and simplicity.

BIOGRAPHICAL SKETCH

Stephanie Weirich is from Farmers Branch, Texas. She received a B.A. from Rice University in 1996 and an M.S. from Cornell University in 2000. Portions of her graduate studies at Cornell were supported by a National Science Foundation Fellowship and an Intel Fellowship. She participated in the Distributed Mentorship Program, sponsored by the Computing Research Association, during Summer 1996 and in the internship program at Bell Labs, Lucent Technologies during Summer 1999.

Stephanie is married to Steve Zdancewic and in Fall 2002 they both will join the Computer and Information Science Department at the University of Pennsylvania, in Philadelphia, Pennsylvania.

ACKNOWLEDGEMENTS

This dissertation would not have been written without the support from many people. First of all my family. My parents Wayne and Charlotte Weirich are my inspiration. My husband Steve Zdancewic deserves much credit, for reasons too numerous to list. Steve's parents Arthur and Deborah Zdancewic have encouraged me as long as I have known them.

I have received much guidance from my advisors, official and otherwise. Greg Morrisett and Karl Crary were extremely influential to this dissertation and to my own professional development. However, I have been getting advice for a long time, and I am also thankful for the wisdom of Matthias Felleisen, Devika Subramanian, Keith Cooper, Bob Constable, Dexter Kozen, and John Reppy.

Life in Ithaca would not be fun without a number of people. My officemates Dave Walker and Nick Howe somehow put up with me for four years. Dan Grossman provided me with coffee filters. Mike Hicks taught me everything I know about dynamic linking. James Cheney, Jim Ezick, Neal Glew, Jason Hickey, Riccardo Pucella, Fred Smith, Yan-ling Wang, and Vicky Weissman made Cornell a great place to study and discuss programming languages. Jennifer Bishop, Lyn Millet, Susannah Howe and Amanda Holland-Minkley made Ithaca a great place to discuss books. I have also been fortunate to share Ithaca with Bert Adams, Gary Adams, Tugkan Batu, Adam Florence, Annette Florence, Takako Hickey, Kim Hicks, Timmy Hicks, Jon Kleinberg, Lillian Lee, Tobias Mayr, Tanya Morrisett, Patrick White and Alyson White.

Stephanie Weirich
July, 2002

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This dissertation is about defining operations with types. More specifically, it explores how mechanisms to support run-time type analysis may be added to statically-typed programming languages. In these languages, *types* describe the structure of each data value. Types also characterize operations by describing the types of data on which they may act. *Polymorphic* operations apply to many types of data. Some of these operations use type information about their arguments to guide their execution. Because these operations are defined over metainformation, they are also called *reflective* operations. Another term for them is *polytypic* because they apply to many types of data. In this work, I use the terms *type-indexed* and *type-analyzing* to describe these operations. Many modern applications and systems critically rely on these type-indexed operations. For example:

Generic algorithms A number of generic algorithms are defined by type information and apply to arbitrary data structures.

- To store a value persistently or to transfer it to another machine, an application must convert it to some serial form. *Serialization* converts any data value into a sequence of bytes suitable for persistent storage or transmission. The structure of the data value guides this conversion.
- Structural *equality* determines whether two complex data structures are equivalent by comparing each component.
- Generalizing structural equality to an *ordering* relation provides the ability to store values into a balanced binary tree. This relation is defined by extending the natural ordering on numbers and characters lexicographically through aggregate data structures.
- *Cloning* produces a "deep" or structurally equivalent copy of its argument, providing a way to duplicate data structures.

- Generic *iterators* and *maps* over data structures are also type-indexed operations. These iterators provide a common interface for accessing the elements of the data structure.
- *Reductions* (also called *folds*) traverse data structures and aggregate values from their components. For example, they may return whether a predicate is true for every value stored in the structure, or may add together all integers stored in a data structure.
- *Zipping* combines two data structures into one. For example, two lists of the same length may be combined into a single list containing pairs of elements.
- Type-indexed operations may be application specific. For example, polytypic programming has been used to implement *generic pattern matching* [Jeu95], *term rewriting* [JJ00], *unification* [JJ98], *data compression* [JJ99] and *genetic algorithms* [Ves97].

Extensible Systems If type information is propagated at runtime, running systems may use it for dynamic update. These systems use metainformation for two reasons.

- *Extensible systems use type information to ensure the stability of the running system.* Newly loaded code must be checked to guarantee that it satisfies the requirements of the running system and provides the necessary interfaces. For example, the Common Object Model [COM02] used pervasively in modern applications, treats objects abstractly and provides access to clients through one or more interfaces. All objects must implement the interface `IUnknown`, which provides the function `QueryInterface`, for clients to call at runtime to determine whether the object implements a particular interface.
- *Extensible systems use type information to adapt to new functionality.* The updated service may make new functionality directly available to principals that must communicate it. The most striking example of this process is to expose new functionality by automatically updating the user interface. For example, with JavaBeans [Jav02], a system may examine the interface of a new component to directly provide user-interface control of the component in the form of check boxes, selection lists and buttons.

Garbage collection To implement accurate garbage collection, the run-time system for a programming language must determine what parts of memory represent live data and what parts the collector may reclaim. To accurately trace program data, the collector must know about the types of every data object in memory–it must know which parts of the data represent pointers that refer to other portions of live data. This type information may be tagged on every data object, or it may be passed to the garbage collector as additional arguments [Tol94, AFH94, WA99, WA01, MSS01].

Compiling parametric polymorphism For programming languages that support parametric polymorphism, run-time type analysis is a useful tool for optimization. It is difficult to compile a polymorphic function, because the types of some of its arguments are not known at compile time, so the compiler does not know how much space to allocate. One solution that compilers for polymorphic languages employ is to specialize polymorphic functions at each type at which they are used. For example, the MLton compiler [CFJW00] and templates in C++ [ISO98] implement a version of parametric polymorphism in this manner. However, such type specialization is not always desirable and in many cases, such as in the presence of separate compilation, dynamic loading, or polymorphic recursion, may not be possible.

It is also possible to compile polymorphic code without specializing it. The standard method is to force all values to be the same size, regardless of their type. This approach requires that values be *fully boxed*—if the representation of a value does not fit into one machine word, a pointer to the value is used. There are two drawbacks to this strategy. First, because all values must be boxed, even code that is not polymorphic will run more slowly. Second, it is not easy for such code to interoperate with other languages. *Representation analysis* or compile-time analysis to determine when certain values may be unboxed eliminates some of this overhead [PL91, Ler92, Pou93]. Extending this operation to run time, where the compiled program determines the actual type of polymorphic arguments, allows all boxing to be eliminated [TMC$^+$96, Tar96, Sha97a, CK02].

1.1 Language support for run-time type analysis

Because type-indexed operations are crucial to modern applications and systems, language support for these operations has been an active area of research. Any linguistic mechanism intended to support the definition of these operations must balance expressiveness, safety and simplicity.

- By *expressiveness*, I mean that the user should be able to write type-indexed algorithms in a natural manner. The code should describe concisely (and in a maintainable manner) how the operation depends on type information.

- By *safety*, I mean that the programming language should help the user write correct code. As an approximation to correctness, the language should statically check that programs maintain type invariants—that values are only used with operations appropriate for their type. It should ensure that structural information that guides type-directed operations is accurate.

- By *simplicity*, I mean that the facilities to support type-indexed operations should not be complicated to use, and should not complicate the implementation or the semantics of the programming language.

Why should we add new linguistic mechanisms to define these operations in the first place? The argument for simple programming languages and the fact that systems have been able to implement many of them in traditional programming languages seems to imply that no specialized functionality is necessary. However, traditional languages typically sacrifice either expressiveness or safety in the implementations of these operations.

For example, one implementation of serialization is to create a data structure to represent type information. The arguments to the serialization function are both a value to serialize and an element of this type to describe the type of that value. However, without specialized language support, what ensures that the correct metainformation has been provided to the serializer? For example, if the type information claims that the value to serialize is an integer when it really is a string a run-time error will occur.

Another implementation of serialization may be safe but difficult to use or maintain. It is possible for the user to write separate serialization routines for every type of data. Static type checking can ensure that the correct routine is called. In object-oriented languages, dynamic dispatch even automates the process of providing the correct serialization routine. However, writing all of these routines and maintaining them as the types of data evolve places significant burden on the programmer. Every class must define a serialization method. There is no way to define serialization once, for all types. Basing the serialization operation on the structure of its argument simplifies its maintenance by allowing it to automatically adapt to changes in data representation.

Finally, a third option that is both safe and simple to maintain is for the programming language to implement serialization as a primitive operation. For example, the Java programming language [GJS96] does so with the method `toString` in the `Object` class. However, in that case, the user lacks control over how serialization operates[1] and the ability to define other type-indexed operations.

[1] Overriding `toString` gives the user some flexibility, but there is no uniform way to change how serialization operates.

In many cases, an application may need to modify type-indexed operations. For example, it may wish to employ a method of serialization that better supports compression, specialized file formatting or encryption. Numerical applications may wish to treat equality for floating-point numbers differently. Applications also may need to modify the order that iterators traverse subcomponents in order to more efficiently use memory caches.

Because there is no expressive, flexible and safe way to implement these operations using traditional language features, there have been a number of proposals for language extensions to support the definition of type-indexed operations. In the following, I discuss a number of these approaches and compare the trade-offs they have made with respect to expressiveness, safety and simplicity. By analyzing these systems we may determine the characteristics of an ideal language that supports type-indexed operations. The goal of this dissertation is to determine how we may combine the advantageous features of these systems and avoid their deficiencies.

1.1.1 Previous approach: Dynamically typed languages

Dynamically typed languages seem well suited for naturally expressing many reflective programs. Languages such as LISP [Ste90], Scheme [KCJR98], and Erlang [Arm97] store type information with every data object, permitting easy access to run-time type information.

For example, writing a serialization function in Scheme is straightforward. Scheme includes primitive predicates such as `boolean?`, `number?`, and `char?` for determining the type of each object. The following code fragment implements the serialization function `valtostring` that converts any Scheme value to a string representation.

```
(define (valtostring obj)
  (cond ((null? obj)      "()")
        ((pair? obj)      (tostring-list obj))
        ((boolean? obj)   (if obj "#t" "#f"))
        ((symbol? obj)    (symbol->string obj))
        ((number? obj)    (number->string obj))
        ((string? obj)    (string-append "\"" obj "\""))
        ((vector? obj)    (tostring-vector obj))
        ((procedure? obj) "#<function>")
        ;; branches for additional types elided
        ... ))
```

The conversions `symbol->string` and `number->string` are also primitive to Scheme, translating Scheme's internal representation of symbols and numbers to a readable form. The auxiliary functions `tostring-vector` and `tostring-list` call `valtostring` for each of the components of the aggregate data structure. As in most languages, in Scheme there is no way to serialize functions, so this implementation of `valtostring` returns `"#<function>"` in that case.

Even though Scheme provides a natural way of expressing operations such as `valtostring`, it does so in a language without the support of static type checking. Languages such as Haskell [PH99] and Standard ML [MTHM97] use a type system to describe a class of programs that are guaranteed not to incur certain errors at run time. These guarantees about statically-typed languages are usually summed up by Milner's catch phrase [Mil78]:

> "Well-typed programs don't go wrong."

This guarantee means that if a program is well typed, then executing the program will not cause a run-time error. The set of proscribed errors depends on the language definition. The type system of ML limits how data values may be used based on their types: for example, programs may not access integers as if they were aggregate structures, use floating-point numbers as if they were functions, or call functions with the wrong number arguments. If the type system determines that a program could do one of these operations, then the ML compiler rejects the program.

Scheme, on the other hand, allows all programs that are syntactically well-formed. In order to determine whether Scheme programs are free of these sorts of errors, developers must exhaustively test them on all possible inputs. In practice, this exhaustive testing is not always feasible or even possible.[2] Therefore, in terms of software development, static typing is essential. It provides a method for mechanical verification of basic correctness properties for a given program.

1.1.2 Previous approach: Reflecting types as data

The statically-typed programming language Java [GJS96] provides capabilities both for determining the run-time class of an object (similar to its type) and

[2]It is important to note that Scheme is still considered a type-safe language. With dynamic typing, the assurance the type system provides is fail-stop behavior. Whenever such a run-time error does occur, the program will immediately halt. In languages that are not type safe (such as C [KR88, ISO99] or C++ [Str97, ISO98]) the result of these erroneous operations is not defined and so reasoning about execution is extremely difficult. For example, if a program writes to an array outside of its bounds there is the possibility of changing the value of some other program variable.

for reflecting the structure of that run-time class. The natural question is whether these are sufficient for developing the applications discussed earlier.

Run-time type dispatch is a central feature of object-oriented programming. When an object invokes a method, the identity of the class of the object determines which version of the method to use. Because of subtyping in Java, the actual class of an object may not be known at compile time. For example, even though some variable `x` is an instance of `LinkedList`, it might be assigned the class `Object`. The `instanceOf` operator may discover run-time type of `x`. The cast expression `(LinkedList)x` will check the run-time class of `x` and either do nothing if the cast succeeds, or raise an exception otherwise.

While these operations add an explicit form of dynamic typing to Java, they do not provide the facility for implementing the reflective applications mentioned above. Because this form of run-time type dispatch operates over the *names* of the types, operations over type structure can only be implemented if the mapping between the type name and the type structure is already known. Using Java's facilities to implement serialization is inconvenient at best. Each new class must include a method to implement serialization for that object.

Instead, the approach taken by Java (and previously by other languages such as Amber [Car86] and Cedar/Mesa [Lam83]) is to reflect type information into a data structure. The Java Reflection API [Gre98] allows any object to call the method `getClass` to retrieve metadata describing the structure of the run-time class. The returned object supports operations for determining the fields and methods of the run-time class. For example, a Java serializer implemented with these operations is below.

```
String valtostring ( Object o ) {
   String result = "\"";
   // determine the fields of this class
   Fields[] f = o.getClass( ).getFields( );
   for ( int i=0; i<f.length; i++ ) {
      Class fc = f[i].getType( );
      if ( fc.isPrimitive( ) ) {
         if ( fc == Integer.TYPE ) {
         // access the field of the object,
         // make sure it is an integer,
         // and then convert it to a string.
            result += toStringInt((Integer) f[i].get( o ) );
         } else if ( fc == Boolean.TYPE ) {
          // access the field of the object,
          // make sure it is an Boolean,
```

```
          // and then convert it to a string.
            result += toStringBoolean((Boolean) f[i].get( o ) );
          }  else if ...
          // additional cases for other primitive types
      } else {
        // call function recursively for objects
          result += valtostring ( f[i].get( o ) );
       }
   }
   return result + "\"" ;
}
```

This example relies heavily on Java's treatment of the type `Object` as a dynamic type. With subtype polymorphism, it may be called on any argument. After this code determines the structure of the run-time class of the argument, it accesses the fields of `o` as `Objects` and then checks them at run time to make sure that they are of the correct class.

Reflective programming in Java is similar to reflective programming in Scheme. Even though the user has the ability to determine the run-time class of an object, that information is not reflected into the type system. Instead, a run-time cast must be used to explicitly change the type of the object. This run-time cast is a redundant check that the object is of the correct type.

```
Integer example (Object x) {
  if (x.getClass() == Integer.TYPE) {
    return ((Integer)x) + 3;
  } else return new Integer(0);
}
```

However, because Java must rely on these run-time casts to ensure that type information is used correctly, there is a possibility of a type error at run time. The `ClassCastException` is raised if one of these casts should fail. While using the reflective mechanisms of Java, we have lost the benefits of static type checking that Java provides. For example, the following Java code is statically type correct, although executing it will throw the exception `ClassCastException`.

```
Integer example (Object x) {
  if (x.getClass() == Boolean.TYPE) {
    return ((Integer) x) + 3;
  } else return new Integer(0);
}
```

The problem with Java's support with run-time type analysis is that while one can reflect run-time class information into a data structure in Java, the use of that data structure is not reflected back into the type system. The type system does not maintain the connection between that data structure and type that it represents.

1.1.3 Previous approach: Dynamic types and typecase

An alternative to reflecting types as data and then analyzing that data is to use a specialized expression form to analyze types. In the calculus of Abadi et al. [ACPP91, ACPR95], a type called `Dynamic` hides the actual type of a value in much the same way as the `Object` type of Java. (Similar functionality exists in Modula-3 [CDG+89] and in a proposed extension to ML by Leroy and Mauny [LM91]). However, instead of producing a value to represent the hidden type (with `getClass`) this language uses a term called `typecase` to directly analyze hidden type information.

Just as any Java value may be coerced to the `Object` type, in this language any value `v` of type `t` may be coerced to the `Dynamic` type with the expression `(dynamic v : t)`. The `typecase` operator pattern-matches against the type information stored in a dynamic type, behaving analogously to the pattern-matching `case` expression of SML [MTHM97]. During execution, this type argument is compared to various type patterns and the matching branch is selected, binding the type variables appearing in that pattern within that branch.

For example, using `typecase` we may write `valtostring` as follows. This example is written with a variant of SML syntax. The expression `fun` defines the recursive function `typetostring` with argument `dv` of type `Dynamic` and return type `string`. Using `typecase`, the hidden type of this argument is matched against the types `int`, `string`, function types (`a->b`), product types (`a*b`), and type `Dynamic`. (In this example, the infix function ++ concatenates two strings.)

```
fun valtostring (dv:Dynamic) : string =
   typecase dv of
     (v: int)      => int2String(v)
   | (v: string)  => "\"" ++ v ++ "\""
   | (v: a -> b)  => "#<function>"
   | (v: a * b)   => "(" ++ valtostring (dynamic fst(v):a) ++ ","
                         ++ valtostring (dynamic snd(v):b) ++ ")"
   | (v: Dynamic) => "dynamic " ++ valtostring (v)
```

In each branch, the dynamic value is rebound to a new variable `v` whose type reflects the matching branch. As `typecase` matches the outermost structure of the type argument, the branches for function types and product types bind the pattern variables `a` and `b` to refer to subcomponents of these types. (In a slight departure from SML syntax, I use `a` instead of `'a` for type variables. All type variables will come from the beginning of the alphabet.) For the recursive call to `valtostring` in the product branch, these types are necessary to convert the components to the type `Dynamic`.

Compared to the Java definition of `valtostring`, this version does not require any run-time type casting. In each branch of `valtostring`, the dynamic value is rebound with a new type, determined by the type matched by that branch. Each branch produces a result of type `string`, so `valtostring` has type `Dynamic -> string`.

In this system, as in Scheme, all type information must be attached to a value of that type. Unlike the reflective `getClass` of Java, there is no way to access this information separately. A direct consequence is that we cannot write a function to display all types in this language. The best that we can do is below.

```
fun typetostring (dv:Dynamic) : string =
   typecase dv of
     (v: int)     => "int"
   | (v: string)  => "string"
   | (v: a -> b)  => "#<function>"
   | (v: a * b)   => "(" ++ typetostring (dynamic fst(v):a) ++ "*"
                         ++ typetostring (dynamic snd(v):b) ++ ")"
   | (v: Dynamic) => "Dynamic"
```

The reason is that in order to print a function type `a -> b`, we must have a value of type `a` and a value of type `b` in order to call `typetostring` recursively. However, there is no good way to get a value of either type.

1.1.4 Previous approach: Intensional polymorphism

Intensional polymorphism separates run-time type information from values. Instead of analyzing an implicit type stored with a dynamic value, in Harper and Morrisett's language λ_i^{ML}, `typecase` analyzes an explicit type [HM95].

Because types may be analyzed separately from terms, `typetostring` has a very natural implementation. (This example is written in a variant of SML syntax, where type abstraction and type application are explicitly notated with square brackets.)

```
fun typetostring [a] : string =
    typecase a of
      int      => "int"
    | string   => "string"
    | (b -> c) => "(" ++ (typetostring [b]) ++ "->"
                      ++ (typetostring [c]) ++ ")"
    | (b * c)  => "(" ++ (typetostring [b]) ++ "*"
                      ++ (typetostring [c]) ++ ")"
```

This expression abstracts the type `a` and returns a string. For example, if the argument is an integer type, `typetostring` returns the string `"int"`. For types composed of other types, such as function types and product types, `typetostring` calls itself recursively to produce the strings of those subcomponents.

In λ_i^{ML}, the implementation of `valtostring` analyzes the type argument `a` to produce a serialization function of type `a -> string`.

```
fun valtostring [a] : a -> string =
    typecase a of
      int      => int2string
    | string   => (fn x:string   => "\"" ++ x ++ "\"")
    | (b -> c) => (fn x:(b -> c) => "#<function>")
    | (b * c)  => (fn x:(b * c)  =>
                    "(" ++ valtostring [b] (fst x) ++ ","
                        ++ valtostring [c] (snd x) ++ ")" )
```

The integer branch of `valtostring` returns the primitive operation for converting integers to strings. If the argument to `valtostring` is already a string, the string branch provides a function to wrap the string in quotation marks. When `a` is a function type, the serializer produces the string `"function"`. Finally, when the argument to `valtostring` is a product type, `valtostring` calls itself recursively. In this branch, the type variables `b` and `c` are bound to the types of the first and second components of the product type, which are then used in the recursive call.

Even though this version of `typecase` does not rebind the type of a dynamic variable, the implementation of `valtostring` does not require casting. The typing rules for `typecase` allow the types of each of these functions to vary. In this example, the integer branch is of type `int -> string`, the string branch is of type `string -> string`, the function branch is of type `(b->c) -> string` and the product branch is of type `(b*c) -> string`. Each of these types is an instance of `a -> string`, so the type of `valtostring` is `forall a. a -> string`.

In this language, an existential type `exists a. a` is equivalent to type `Dynamic`. Almost any example that may be written in the previous language may be written in λ_i^{ML}. The only limitation is that while it is legal to wrap a dynamic value twice, i.e. `(dynamic v : dynamic)`, λ_i^{ML} is based on *predicative* polymorphism, so it cannot hide an existential type.[3]

However, because of the separation between type information and values that λ_i^{ML} allows, this `typecase` is more expressive. This language is built on *parametric polymorphism* [Gir72, Rey83], as in Haskell or ML, so it has the ability to express that types are equal. For example, if `x` and `y` both have type `a`, then we know that they have the same type even if we do not know what it is. If `x` and `y` have type `Dynamic`, then their actual types may be unrelated to each other. Because we can express such type equalities, when the identity of the abstract type `a` is determined, the type system may connect that knowledge to all terms of type `a`.

Other similar frameworks to intensional polymorphism include extensional polymorphism [DRW95], structural polymorphism [Rue92, Rue98], and type-parametric programming [She93].

1.1.5 Previous approach: Polytypic programming

The approaches in the preceding sections share one deficiency. Type-indexed operations may be defined only over closed types.

Many type-indexed operations must be defined over parameterized types. For example, compare the function `listlength`, of type `list a -> int`, that counts the number of `a`'s occurring in a given list, to the related function `treelength`, of type `tree a -> int`, that counts the number of `a`'s occurring in a given tree.[4] Both of these functions are instances of a type-indexed function `length`. This function `length` is defined over parameterized types, such as `list` and `tree` instead of closed types, such as `list int` or `tree bool`.

Various systems of *polytypic* or *generic* programming provide functionality to define such operations as `length`. For example, the programming language *Charity* [CF92] automatically defines maps and catamorphisms for each user-defined datatype through a systematic encoding in the polymorphic lambda calculus [BB85]. Jansson and Jeuring's PolyP extension of Haskell [JJ97, Jan00] supports a wide variety of operations such as catamorphisms, maps and zipping functions. Bellé, Jay and Moggi's FML supports functorial polymorphism [Jay95, JBM98]. Every parameterized type is one component of a functor. The other component of this functor is a mapping operation for that parameterized type, which may be used to define other polytypic operations. Hinze's system defines polytypic functions by *interpreting* language of types in the term

[3] An extension of λ_i^{ML} with a impredicative polymorphism [TSS00] removes this restriction.

[4] In another departure from SML syntax, I write parameterized types with prefix application. For example a list of integers has type `list int` instead of `int list`.

language [Hin00]. Hinze and Peyton Jones use this framework to extend the automatic derivation of Haskell type classes [HJ00]. This framework is also the basis of the Generic Haskell compiler [CHJ+01].

The chief concern of polytypic programming has been with how to define various type-indexed operations. Therefore, the implementations of these systems rely on the compiler to generate specialized code for each type of data, based on these definitions. None of these systems support run-time type information or analysis.

For a number of reasons, run-time type analysis is an important part of a system supporting type-indexed operations. Specializing polytypic functions requires making a copy of the function for each instantiation, leading to an increase in the size of the resulting program. Furthermore, in many languages it is not possible to specialize polytypic functions. Languages that support separate compilation or dynamic loading cannot specialize polytypic functions, as not all applications of a function may be known at compile time. Languages that support polymorphic recursion (such as Haskell [PH99]) also cannot specialize polytypic functions because each recursive call of the function may be instantiated at a new type.

1.2 An ideal language

None of these approaches provides a completely satisfactory system for implementing type-directed operations. However, by examining these previous approaches, we may identify the important features of a language that supports such operations.

1. *Programs using type information should be statically type checkable.* Static type checking is an essential part of program development. The mechanisms for defining type-indexed operations should have the same static guarantees as the rest of the language.

2. *The language should be based on the polymorphic lambda calculus.* Except for Scheme, which has no static type system, and Java, all of the previous approaches in the last section were based around languages with parametric polymorphism. (Furthermore, there are many proposals to add such polymorphism to Java [MBL97, AFM97, BOSW98, CS98].) This fact is no accident—parametric polymorphism, also called *generics*, is essential to the expressiveness of a typed programming languages.

   The Girard-Reynolds polymorphic lambda calculus [Gir72, Rey83] describes a very strong form of parametric polymorphism. In the past, various languages have restricted the polymorphism that it provides for various reasons.

For example, λ_i^{ML} restricts polymorphism to a *predicative* form [ML75]. Polymorphic types do not quantify over other polymorphic types. However, this restriction forces polymorphic functions and existential packages to become "second-class citizens"—they may not be used in the same way as other functions and data structures. Because modern programming paradigms require that we manipulate abstract data structures, we cannot accept this restriction.

Language such as ML and Haskell are *impredicative* but they restrict Girard-Reynolds polymorphism in other ways in order to support complete type inference [TU96, Wel99]. This type system, defined by Hindley-Milner type inference [Mil78], forbids features that cause trouble: first class polymorphism, polymorphic recursion and higher-order polymorphism.

- *First-class polymorphism* means that polymorphic functions and existential packages may be passed to functions and stored in data structures. Prohibiting first-class polymorphism means that the type of all polymorphic functions must have the quantifiers at the top level.
- *Higher-order polymorphism* means that functions may quantify over type constructors as well as types. This expressiveness is necessary to support parameterized data structures.
- *Polymorphic recursion* means that a recursive polymorphic function may be instantiated at a different type at its recursive call. *Nested datatypes* [BM98, Oka99] and other expressive functional data structures require polymorphic recursion.

It is rapidly becoming apparent that the expressiveness provided by these features will be an essential part of next generation programming languages. Already, many languages compromise full type inference to allow some of this functionality [OL96, Jon97, PT98]. Furthermore, including all of these features *simplifies* the type system of an annotated programming language because it eliminates restrictions made to support type inference.

3. *Type information should exist at run-time.* Type-indexed operations should be available for types that are unknown at compile time so the language may support separate compilation, dynamic loading and polymorphic recursion.

4. *Type information should be independent of values.* Dynamic typing systems such as Java and Scheme (and the type `Dynamic` calculus to some extent) require that every value have associated run-time type information. Furthermore, each piece of type information is allowed to describe the type of only

one value. This framework makes it difficult to express some type-indexed operations and makes it difficult to optimize the use of run-time type information.

5. *Run-time type information should be separate from compile-time type information.* In a typed programming language, it is customary for types to describe only terms and for terms to model all computation. Yet, in languages such as λ_i^{ML}, computation depends on the type information. This dependence means that types play two roles—they both describe terms and they exist at run time. In λ_i^{ML} these two roles behave similarly as both are modeled by lambda calculi. However, mechanisms that more precisely describe computation, such as effect systems and allocation semantics, will complicate type checking if they must also describe the run-time behavior of types.

6. *The mechanisms for run-time type analysis should be easy to incorporate.* An advantage of the Java approach was that it was easy to incorporate into the language. In particular, the components for supporting type-indexed operations did not require new typing rules so they did not change the soundness properties of the language. In order for any mechanism that define type-indexed operations to be successful, it must not require complicated proofs or implementation.

7. *All type information should be analyzable.* Many systems do not support the analysis of types with binding structure (such as polymorphic types). In order to have a complete and expressive system, it is important that all types may be examined. Furthermore, many typed programming languages include language elements in the type system that do not describe terms. These *type constructors*, similar to the functions, products and sums of the term language, describe the relationship between types. Many polytypic algorithms, such as iterations, maps, reductions and zips must be defined by such type constructors.

8. *It should allow the definition of new types.* A programming language may allow the user to create new types so that compile-time type error messages can be application specific. With Haskell type classes [WB89, HHJW96], one can easily extend a type-directed operation with these new types merely by creating a new instance of that class. This mechanism allows type-indexed operations to have application specific behavior.

9. *It should support type-level type analysis.* Because type-directed operations often require a type translation to describe the result type, there must be

Table 1.1: Languages described in this dissertation

Chapter	Name	Description
2	LI	Intensional type analysis, based on λ_i^{ML} [HM95].
3	LIR	Type information fully reflected into term language, allowing type erasure, based on λ_R [CWM98].
4	LX	Designed to support the compilation of type-analyzing code [CW99a].
	LXR	As above, but permits type erasure [CW99a].
5	LU	A version of the polymorphic lambda calculus, derived from Girard's λU^- [Gir72].
6	LH	Extends run-time analysis to higher-order types [Wei02].
7	LK	An alternate version of LH.
	LKR	As above, but permits type erasure.

some mechanism at the type level to describe how types depend on other types. The *Typerec* type constructor of λ_i^{ML} is an example of such a facility.[5]

1.3 Dissertation outline and contributions

Inspired by the features listed in the previous section, I intend this dissertation to answer the following questions.

1. How can we distinguish between compile-time types and run-time type information in a type safe manner?
2. What linguistic constructs are necessary in a programming language to support run-time type analysis?
3. How can we generalize type analysis to include the run-time analysis of all type information?

[5] I describe this type constructor in more detail in the next section.

In answering these questions, I will present a number of different typed lambda calculi, each modeling a different form of run-time type analysis. For reference, Table 1.1 lists the chapters in which each language is described. Why does this list include so many languages? It is in the comparison between these languages that the answers to the above questions reside. By bringing them together for the first time in one work, I hope to make these comparisons more apparent.

To lay the foundations for this work, Chapter 2 starts with a version of Harper and Morrisett's λ_i^{ML} [HM95] that I call LI. The difference between this calculus and λ_i^{ML} are minimal. The changes I have made allow a simpler development of the languages described in later chapters. In this chapter, I also use LI to demonstrate several concrete examples of the type-directed operations mentioned earlier. These examples include a description of how to implement dynamic types in LI that originally appeared in the 2000 International Conference on Functional Programming [Wei00].

Chapter 3 provides a simple answer the first question by presenting a version of the λ_R language of Crary, Weirich and Morrisett [CWM98]. In this chapter, I describe how type-directed execution may be performed without depending on type-level information at run time. The key idea is that analysis of types may be simulated by analysis of terms that stand in for the type arguments. This process works if these term representations have a form of *singleton type* that describes what type they represent. This chapter describes the language LIR that includes term representations of types and a special type constructor R. I end this chapter with an embedding of LI to LIR that also appears in the Journal of Functional Programming [CWM02].

Chapter 4 presents the LX language of Crary and Weirich [CW99a], which includes functions, products, sums and a restricted form of iteration in the type constructor language. Instead of determining the identity of an unknown type as in LI, this language provides an operator to determine the identity of an unknown constructor sum. With the facility to analyze unknown constructor sums, the LX language may describe the analysis of a wide variety of static information, and so provides an initial answer to the third question.

With this new form of analysis, we have more flexibility. In contrast to LI, where the analyzed type system is specifically hard-wired into the language, in LX, one may program at the type-level to encode the type system in the constructor language. An application of this flexibility is in the area of typed compilation. During the compilation of a type-analyzing language, analysis of high-level types must be represented in the low-level language. In LX, the high-level types may be encoded with inductive sums of products. This facility also means that the LX language is the first to support the type-level analysis of quantified types. Because the separation in LIR between the static language and the dynamic language is so important, in the second part of this chapter, I describe Crary and Weirich's erasure version, which I call LXR, and our translation between LX and LXR. The LX and LXR languages and the translation between them originally appeared in the 1999 International Conference on Functional Programming [CW99a].

Inspired by the encoding of type systems with the LX language, Chapter 5 presents an answer to the second question. The LXR language is quite complicated, with products, sums and inductive types used to encode the type system of a language like LI. This type system has the structure of an inductive datatype. It is also possible to encode such datatypes with the polymorphic lambda calculus. Using this idea, this chapter presents an encoding of LIR into LU, an extension of F_ω with kind polymorphism. Therefore, sophisticated machinery does not need to be incorporated into the semantics of a language in order to support type-analysis, only sufficiently rich polymorphism. The translation presented in this chapter originally appeared in the 2001 European Symposium on Programming [Wei01].

In Chapter 6, I return to the third question and consider type functions. The facilities of LI cannot be used to define operations over parameterized types, such as generic iterators. Therefore, this chapter extends LI to the LH language, with support for lifting type analysis through higher-order types. Furthermore, because quantified types may be represented with higher-order abstract syntax, this chapter also provides insight into their analysis. This language was originally published in the 2002 European Symposium on Programming [Wei02]. Finally, in Chapter 7, I demonstrate that this functionality is still amenable to development as a type-erasure language, in the sense of LIR.

1.4 Reflection

Analyzing types at run time is also called *reflection* because run-time computation is defined over metainformation: the structure of data values. However, the term reflection refers to many ideas in many contexts, and so I briefly mention those alternative meanings in this section. I do not intend to fully cover this broad and diverse topic—several survey papers [DM95, Har95] provide a more thorough introduction to this area.

In general, reflection usually refers to a programming language reading and modifying its current state. Smith's seminal papers [Smi84, dRS84] on *computational reflection* formalized reflection in programming languages with the languages 2-LISP and 3-LISP. In these languages, reflection was implemented through an arbitrary number of interpreters each manipulating the code and data of the language at a different level. Computational reflection has also been studied by Maes [Mae87] and Wanatabe and Yonezawa [WY88]. The Java Reflection API is the product of a large study of reflection in several object-oriented languages such as Smalltalk [Coi87] and CLOS [KdRB91]. Current research on Aspect-Oriented Programming [KLM+97] extends the capabilities of Java reflection.

Even before programming languages, reflection was already studied in the context of logical systems. In logic, reflection refers to a formal system reasoning about itself. It may do so by encoding the formulas, axioms, inference rules and proofs of that formal system with its own elements, such as the integers. For example, the system might have a map $\ulcorner\cdot\urcorner$ between formulas and integers, and a proposition $\mathit{proves}(p, \ulcorner\phi\urcorner)$ which states that p is a proof of the representation of the formula ϕ.

Reflection was a key part of Gödel's incompleteness theorem [Göd31]. By constructing a formula that asserted its own lack of proof, Gödel was able to show that no system expressive enough to include reflection could be both complete and consistent. Because Gödel's theorem talks about the limitations of formal systems, it also shows how those formal systems may be strengthened.

Feferman [Fef62] introduced the term *reflection principle* as "the description of a procedure for adding to any set of axioms A certain new axioms whose validity follows from the validity of the axioms A and which formally express, in the language of A, evident consequences of the assumption that all theorems of A are valid." For example, adding a new axiom about the consistency of A produces a stronger system than before.

Alternatively, reflection may be added to a formal system not to strengthen it, but to make proofs shorter. It amounts to adding an axiom such as the following to the formal system, allowing formally proved properties as additional axioms.

$$\vdash \phi \text{ if } \exists p.\mathit{proves}(p, \ulcorner\phi\urcorner)$$

Variants of this *reflection rule* have been studied in the context of the theorem prover Nuprl [ACHA90]. Here, this rule says that to prove a goal G under hypothesis H, it suffices to show that a representation of the statement H implies G is provable. Care must be taken to avoid developing an inconsistent system. In order to prove the soundness of this rule, one must assume the soundness of Nuprl with this rule, as it may have been used in the proof of $H \vdash G$. However, Nuprl is not sound unless this rule is sound. The solution is stratification: parameterize this rule by its reflection level, and require that encoded sequents only use this rule at a lower level.

Chapter 2

Background: A calculus for dynamic type dispatch

This chapter presents a slightly modified form of Harper and Morrisett's λ_i^{ML} calculus [HM95, Mor95] (called LI) in order to provide an initial framework for intensional type analysis. The λ_i^{ML} language is quite expressive; it and its derivatives were used extensively in a number of high-performance ML compilers including TIL/ML [TMC$^+$96, MTC$^+$96], FLINT [Sha97b] and TILT [PCHS00]. Section 2.1 informally demonstrates the use of type analysis in this language, with the goal of providing an intuition about how the type-indexed operations described in the introduction may be expressed. The technical material of this chapter begins in Section 2.2.2 with a brief explanation of the syntax of LI and a brief catalog of the elements of the core language. This core language will be the basis of all of the languages in the following chapters. Section 2.3 covers the semantics of *typerec* and *Typerec*, the specific operators of LI supporting type analysis. Finally, the last section of the chapter lists several properties of the LI language and sketches their proofs.

2.1 Examples of type analysis

To begin, I start with a few informal examples of intensionally polymorphic functions. Each of these functions may be expressed in LI, but for initial presentation, these examples follow in a modified version of Standard ML (SML) syntax [MTHM97].

In SML, the types of each expression may be automatically inferred by the type checker. To make the examples more clear, however, I annotate each function with the types of its arguments. Furthermore, although SML does not explicitly express type abstraction and instantiation, in these examples, it is important to know where they occur. In SML, the keyword `fun` creates a recursive definition, which may be parameterized by one or more term variables. In the following examples, I also allow recursive definitions to be parameterized by explicit type variables, notated with square brackets. Furthermore, all type instantiations will be explicit, again in square brackets.

2.1.1 Data representation

An important application of type analysis is for efficient data representation by a typed intermediate language. More examples of these techniques in the domain of typed-directed compilation appear in the work of Morrisett [Mor95].

Suppose we want to store an array of Boolean values. Most computer architectures require that memory accesses are a word at a time, but it is a waste of space to store Booleans as integers. A solution is to pack thirty-two Booleans into one word and use bit manipulations to retrieve the correct value. To subscript from a packed Boolean array, we might use the following function (with `<<` for shift left, `&` for bitwise and, and `<>` for inequality):

```
fun bitsub (a:array int, i:int) : bool =
     sub(a,i div 32) & (1<<(i mod 32)) <> 0
```

This function is fine when we know a given array contains Boolean values, but we would like code polymorphic over all arrays to be able to use this mechanism. In other words, we would like to write a polymorphic array subscript that employs `bitsub` when the array is an array of bits and uses the default subscript operation on arrays of other types.

```
fun packedsub[a] =
   typecase a of
     bool => bitsub
   | _    => sub
```

But what is the type of `packedsub`? It is not `(array a * int) -> a` as we might expect because the type of `bitsub` is `(array int * int) -> int` when `a` is `bool`. Just as the operation of `packedsub` depends on the type `a`, the *type* of `packedsub` also depends on the type `a`. Therefore, we need an additional form of intensional type analysis to describe this type.

The construct `Typecase` describes how *types* can be produced by analysis of other types. Using `Typecase`, we can define a type of array (called a `PackedArray` below) that will produce an array of integers to hold Booleans and an ordinary array for other types.

```
type PackedArray[a] =
   Typecase a of
     bool => array int
   | _    => array a
```

With this definition, the type of `packedsub` is `(PackedArray a * int) -> a`.

2.1.2 Polymorphic equality

```
type Eq a =
   Typecase a of
     int      => int
   | (b -> c) => void
   | (b * c)  => (Eq b) * (Eq c)

fun eq[a] : Eq a -> Eq a -> bool =
   typecase a of
     int      => primIntEq
   | (b -> c) => (fn x => fn y => void)
   | (b * c)  =>
       (fn (x,y) => fn (w,z) =>
          eq[b] x w andalso eq[c] y z)
```

Figure 2.1: Example: Polymorphic equality

Another typical use of `Typecase` is to restrict the domain that a type-directed function may be called. This use is analogous to Haskell type classes [WB89], which define a predicate over types indicating the members of the type class. For example, in Haskell one would declare that the equality class supports a method `eq` implementing polymorphic equality.

```
class Eq a where
   eq :: a -> a -> Bool
```

Next, one would declare the instances of `Eq`. Below integers and tuples are members of the equality class. Instances also include the definition of the member function at that type.

```
instance Eq Int where
   eq = primIntEq
instance (Eq a, Eq b) => Eq (a,b) where
   eq (x,y) (w,z) = eq x w andalso eq y z
```

The lack of an instance declaration for function types means that they are not part of the class—polymorphic equality is not defined for them. With type analysis, to restrict the domain of polytypic functions, we use `Typecase` to write an almost identity function, replacing types which are not members of the domain (such as function types) by the type *void*. As there are no elements of type *void*, polymorphic equality can never be called on function arguments.

2.1.3 Run-time type checking and dynamic types

In order to call the polymorphic equality function of the previous section, we must show that type of the first argument is an equality type, of the form `Eq a`. However, if we cannot prove this fact statically, is there no way that we can call the equality function? What if all we know is that the two objects are of type `a`? How can we determine if `a` is an equality type?

Because we have the ability to determine the identities of abstract types at run time, we can compare `a` with `Eq a` and determine if they are the same. We can write a function of type $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$ to convert an object of type `a` to type `Eq a` if the types match. By checking the types at run time, we have the ability to circumvent the type system. Even though there is no way to prove the equivalence of two types statically, we have the ability to include a dynamic proof. With this power comes extra responsibility: the dynamic proof could fail. If `a` is instantiated with a function type, then `Eq a` will be `void` and the cast will fail because the types do not match. A programmer using this function must handle the error case and provide an alternative if it should occur.

Below I present two versions of the function *cast* taken from Weirich [Wei00]. If the type arguments match, cast will just return its term argument at the new type; otherwise it will raise an exception.[1]

An initial implementation of `cast` appears in Figure 2.2. In the first branch of the `typecase`, `a` and `b` have been determined to both be to `int`. Casting an `int` to an `int` is just an identity function.

In the second branch of the `typecase`, both `a` and `b` are product types (`a1 * a2` and `b1 * b2` respectively). Through recursion, we can cast the subcomponents

[1] It would also be reasonable to produce a function of type `a -> (option b)`, but checking the return values of recursive calls for `NONE` only lengthens the example.

```
fun cast[a][b] : a -> b =
   typecase (a,b) of
     (int,int)           => (fn x => x)
   | (a1 * a2, b1 * b2)  =>
        (fn (x:a1 * a2) =>
            (cast[a1][b1] (fst x), cast[a2][b2] (snd x)))
   | (a1 -> a2, b1 -> b2) =>
        (fn (x:a1 -> a2) =>
            cast[a2][b2] o x o cast[b1][a1])
   | (_,_) => error CantCast
```

Figure 2.2: Example: *cast* (Version 1)

of the type (`a1` to `b1` and `a2` to `b2`). Therefore, to cast a product we break it apart, cast each component separately and then create a new pair.

The code is a little different for the next branch, when `a` and `b` are both function types, due to contravariance. Here, given `x`, a function from `a1` to `a2`, we want to return a function from `b1` to `b2`. We can apply cast to `a2` and `b2` to get a function, `g`, that casts the result type, and compose `g` with the argument `x` to get a function from `a1` to `b2`. Then we can compose that resulting function with a reverse cast from `b1` to `a1` to get a function from `b1` to `b2`. Finally, if the types do not match we raise the exception `CantCast`.

However, there is a problem with this solution. Intuitively, all a cast function should do at run time is recursively compare the two types. Unfortunately, unless the types `t1` and `t2` are both `int`, the result of `cast[t1][t2]` does much more. Every pair in the argument is broken apart and remade, and every function is wrapped between two instantiations of cast. This operation resembles a virus, infecting every function it comes in contact with and causing needless work for every product.

The reason we had to break apart the pair in forming the coercion function for product types is that all we had available was a function (from `a1 -> b1`) to coerce the first component of the pair. If we could somehow create a function that coerces this component while it was still a part of the pair, we could have applied it to the pair as a whole. In other words, we want two functions, one from `(a1 * a2) -> (b1 * a2)` and one from `(b1 * a2) -> (b1 * b2)`.

Motivated by the last example, we want to write a function that can coerce the type of *part* of its argument. This function will allow us to pass the same value as the `x` argument for each recursive call and only refine part of its type. We

```
fun cast'[a][b][c] : (c a) -> (c b) =
   typecase (a,b) of
     (int,int)           => (fn x => x)
   | (a1 * a2, b1 * b2)  =>
            (cast'[a1][b1][fn d => c (d * b2)]) o
            (cast'[a2][b2][fn d => c (a1 * d)])
   | (a1 -> a2, b1 -> b2) =>
            (cast'[a1][b1][fn d => c (d -> b2)]) o
            (cast'[a2][b2][fn d => c (a1 -> d)])
   | (_,_) => error CantCast

fun cast[a][b] : a -> b =
   cast' [a][b][fn d => d]
```

Figure 2.3: Example: *cast* (Version 2)

cannot eliminate `x` completely, as we are changing its type. Since we want to cast many parts of the type of `x`, we need to abstract the relationship between the type argument to be analyzed and the type of `x`.

The solution in Figure 2.3 defines a helper function `cast'` that abstracts not just the types `a` and `b` for analysis, but an additional type constructor argument `c`. When `c` is applied to the type `a` we get the type of `x`, when it is applied to `b` we get the return type of the cast. For example, if `c` is `(fn d => d * a2)`, we get a function from type `a * a2` to `b * a2`. We initially call `cast'` with the identity function.

With the recursive call to `cast'`, in the branch for product types we create a function to cast the first component of the tuple (converting `a1` to `b1`) by supplying the type constructor `(fn d => c (d * a2))` for `c`. As `x` is of type `c (a1 * a2)`, this application results in something of type `c (b1 * a2)`. In the next recursive call, for the second component of the pair, the first component is already of type `b2`, so the type constructor argument reflects that fact.

Surprisingly, the branch for comparing function types is analogous to that of products. We coerce the argument type of the function in the same manner as we coerced the first component of the tuple— calling `cast'` recursively to produce a function to cast from type `c (a1 -> a2)` to type `c (b1 -> a2)`. A second recursive call handles the result type of the function.

2.1.4 Reflecting functions

```
datatype exp = Var   of string         (* Variable          *)
             | Const of int            (* Constant          *)
             | Fun   of (string * exp) (* Function          *)
             | App   of (exp * exp)    (* Application       *)
             | Pair  of (exp * exp)    (* Product           *)
             | Pi1   of exp            (* First projection  *)
             | Pi2   of exp            (* Second projection *)

fun reify[a] : exp -> a =
  typecase a of
    int      => fn (Const i) => i
  | (b -> c) =>
       (fn f:exp => (reify[c]) o (App f) o (reflect[b]))
  | (b * c)  =>
       (fn p:exp => (reify[b] (Pi1 p), reify[c] (Pi2 p)))
and reflect[a] : a -> exp =
   typecase a of
     int      => Const
   | (b -> c) =>
       (fn f:(b->c) =>
         let s = gensym ()
         in Fn (s, reflect[c] (f (reify [b] (Var s)))) end)
   | (b * c)  =>
       (fn p:(b*c) =>
         Pair( reflect[b] (fst p), reflect[c] (fst p)))
```

Figure 2.4: Example: Normalization by evaluation

While the goal of structural reflection is to provide complete access to the state of the program currently executing, few languages or systems actually provide mechanisms for reifying functions, or creating a representation of program code. Representing computations as well as values is a difficult task. Unlike other sorts of data, such as integers, tuples and arrays where the run-time representation of the data is easy to understand, most language implement first-class functions as a pointer to some piece of compiled code. Providing access to this binary data is in itself not very useful. The translation between this compiled binary and the abstract syntax of the programming language is complicated and not uniform over various machine architectures.

However, sometimes it is possible to produce a representation of a computation, if that computation is written in a typed lambda calculus and the type of that computation is known. Figure 2.4 contains a datatype for representing the abstract syntax for such a language. The process of producing an element of this datatype from a first class function is reflection. Translating it back into a term is reification. This technique is known as *normalization by evaluation* [BS91] because the reflected version represents the *β-normal, η-long* form of the term. For example:

```
> reify[exp -> exp] (fn x => x)
Fn ("X", Var "X")
> reify[(exp * exp) -> exp] (fn (x,y) => x)
Fn ("X", Pi1 (Var "X")
```

Because reification followed by reflection computes the normal form of the term, this technique has been employed in the area of partial evaluation [Dan96]. In this optimization, if some of the arguments to a multi-argument function are the same at all uses of the function, it makes sense to optimize a version of the function specialized to those arguments. The hard part is specializing the function at run time without the source code. However, applying the function to those arguments and then normalizing the function will produce such an optimized version.

2.2 The LI language

Type analysis in the LI language is called *intensional analysis* of type information. The term intensional analysis refers to the fact that types are analyzed by their internal structure as opposed to their extensional properties, or with respect to the terms that they contain. The intensional operators in this language follow earlier work by Constable [Con82, CZ84]. In this next section, I describe a formal language containing an operator very similar to `typecase`.

2.2.1 LI Syntax

The LI language contains four syntactic classes: at the lowest level, *terms* model evaluation of the language at run time and are described by *types*. The language of *type constructors* can be used to “compute” types (or express relationships between types in a functional notation) and are themselves described by *kinds* much in the same way that terms are described by types. Certain type constructors (those of the kind “Type” written $\star$) correspond exactly to the types and the injection

Table 2.1: LI: Syntax

(kinds)	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2$
(constructors)	c	$::=$	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2$
		$\mid$	$\mathit{int} \mid \to \mid \times$
		$\mid$	$\mathit{Typerec}\ c\,(c_{\mathit{int}}, c_{\to}, c_{\times})$
(types)	σ	$::=$	$T(c) \mid \mathit{int} \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2$
		$\mid$	$\forall\alpha{:}\kappa.\sigma \mid \exists\alpha{:}\kappa.\sigma$
(terms)	e	$::=$	$i \mid x \mid \lambda x{:}\sigma.e \mid \mathit{fix}\ f{:}\sigma.v \mid e_1 e_2 \mid \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e$
		$\mid$	$\Lambda\alpha{:}\kappa.e \mid e[c]$
		$\mid$	$\mathit{pack}\ \langle c, e\rangle\ \mathit{as}\ \exists\alpha{:}\kappa.\sigma \mid \mathit{unpack}\langle\alpha, x\rangle = e_1\ \mathit{in}\ e_2$
		$\mid$	$\mathit{typerec}[\alpha.\sigma]\ c\ \ \mathit{of}\ e_{\mathit{int}}\ e_{\to}\ e_{\times}$
(values)	v	$::=$	$i \mid \lambda x{:}\sigma.e \mid \mathit{fix}\ x{:}\sigma.v \mid \langle v_1, v_2\rangle$
		$\mid$	$\Lambda\alpha{:}\kappa.e \mid \mathit{pack}\langle c, v\rangle\ \mathit{as}\ \exists\alpha{:}\kappa.\sigma$

$T()$ witnesses this correspondence. I will use the word "type" to refer to both the syntactic category of types and to those elements of the type constructor language of kind $\star$, when the distinction is unimportant. In later languages of this dissertation, these two categories will be combined.

The distinguishing features of LI are the type analysis operators *Typerec* and *typerec* in the type constructor and term languages. *Typerec* describes how a type may depend on another type, while *typerec* describes how a term may depend on a type. This language is stratified so that the quantified types $\forall\alpha{:}\kappa.\sigma$ and $\exists\alpha{:}\kappa.\sigma$ range only over type constructors. Because no type constructors correspond to these quantified types, LI has a predicative form of polymorphism [ML75]. This stratification also serves another purpose: it ensures that the arguments to *Typerec* and *typerec* are inductive. Closed type constructors will always be equivalent to one of the monotypes, *int*, arrow or product types.

2.2.2 Core language

As a gentle introduction to the notations employed in this work, I will first describe the elements of the core language of LI (those not involved in type analysis) and define their static and dynamic semantics. Those readers familiar with typed languages and their semantics may wish to skip ahead to Subsection 2.3, where I describe the semantics of the type analysis operators of LI. While the notation used in this section is slightly non-standard, the elements described are common to many typed programming languages. Background material on this core language may be found in a number of sources [Bar92, Mit96, Har01, Pie02].

Table 2.2.1 shows the abstract syntax of LI, listing its four syntactic classes. For the purposes of substitution, it is important to distinguish between free and bound variables. In type constructors, the variable α is bound within c in the form $\lambda\alpha{:}\kappa.c$. In the types, α is bound within σ in $\forall\alpha{:}\kappa.\sigma$ and $\exists\alpha{:}\kappa.\sigma$. Finally, in terms, x is bound within e in $\lambda x{:}\sigma.e$, α is bound within e in $\Lambda\alpha{:}\kappa.e$, α and x are bound within e_2 in $\mathit{unpack}\langle\alpha, x\rangle = e_1 \mathit{in}\, e_2$ and α is bound within σ in $\mathit{typerec}[\alpha.\sigma]\ c\ \mathit{of}\ \ e_{\mathit{int}}\ \ e_{\rightarrow}\ \ e_{\times}$. All variables that are not bound are considered free. The notation $c_1[c_2/\alpha]$ refers to the capture-avoiding substitution of c_2 for the free variable α in the constructor c_1, likewise $e_1[c_1/x]$ is capture-avoiding substitution in terms. In this substitution, if a free variable in c_2 has the same name as a bound variable in c_1, it is possible that substitution could incorrectly bind that free variable. Therefore, during substitution, the bound variables of c_1 must be renamed so they do not conflict with the free variables of c_2. We adopt the Barandregt variable convention [Bar84], identifying all terms that are α-equivalent. Two terms are α-equivalent if they differ only in the names of their bound variables.

With the exception of the kinds, not all productions from this context-free grammar are meaningful. Therefore, to decide what constructors, types or terms are *well formed*, we use a set of typing judgments, or relations between the syntactic classes that declare when an expression of the abstract syntax is well formed. For each of these judgments, there is a set of inference rules allowing one to conclude that judgment. Some of these inference rules have no preconditions: these are the axioms. For example, the judgment $\Delta \vdash c : \kappa$ reads "in context Δ, the constructor c is well formed and of kind κ." The axiom

$$\frac{}{\Delta \vdash \mathit{int} : \star}$$

declares that we may always derive that the constructor *int* is well formed and of kind $\star$ in any context. For the other rules, the judgments in the precondition of the rule (above the horizontal line) must be derived before the judgments below the lines may be concluded. In this way, we may produce a derivation that an expression is well formed. For example, the following derivation that the constructor $(\times)\ \ \mathit{int}\ \ \mathit{int}$ is well formed uses several rules from Table 2.3. (This constructor is

Table 2.2: LI: Judgment forms

Judgment	Meaning
$\Delta \vdash c : \kappa$	c is a valid constructor of kind κ
$\Delta \vdash \sigma$	σ is a valid type
$\Delta \vdash c_1 = c_2 : \kappa$	c_1 and c_2 are equal constructors
$\Delta \vdash \sigma_1 = \sigma_2$	σ_1 and σ_2 are equal types
$\Delta; \Gamma \vdash e : \sigma$	e is a term of type σ

equivalent to the type $int \times int$.)

$$\dfrac{\dfrac{\overline{\Delta \vdash \times : \star \to \star \to \star} \qquad \overline{\Delta \vdash int : \star}}{\Delta \vdash (\times)\ \ int : \star \to \star} \qquad \overline{\Delta \vdash int : \star}}{\Delta \vdash (\times)\ \ int\ \ int : \star}$$

For readability, I will write application of the constructors $\to$ and $\times$ using infix notation, when applied to two arguments (as will almost always be the case). For example, instead of writing $\times\ \ int\ \ int$, I will write $int \times int$.

The forms of the static judgments are shown in Table 2.2. Because I will cover a number of languages with similar judgment forms, I will use $\vdash_i$ to indicate that the judgment (or rule) is specifically for the LI language when it is not clear from context. All of these judgments include the presence of a context Δ or context $\Delta; \Gamma$. The context Δ is finite map between type constructor variables, α, and their kinds, and Γ is a finite map from term variables, x, and their types. The syntax $\Delta(\alpha)$ and $\Gamma(x)$ retrieves the associated kind or type. A map Δ or Γ may be extended with new mappings by the notations $\Delta, \alpha{:}\kappa$ or $\Gamma, x{:}\sigma$. In all cases, we assume that the new variable is not already in the domain of the map. The notation $\Gamma[\alpha/c]$ denotes the (capture-avoiding) substitution of the constructor c for each occurrence of α in the types of the term variables in Γ. The symbol $\emptyset$ explicitly refers to an empty context, but is often omitted.

The core type constructor language is the simply-typed lambda calculus, augmented with a number of constants used to form types. The formation rules for constructors (Table 2.3) are standard. Variables must appear in the enclosing context. Constructor functions $\lambda\alpha{:}\kappa.c$ may be created if their bodies are well formed under a context extended with the bound variable. Constructors may be applied if they are of function kind ($\kappa_1 \to \kappa_2$) and the actual argument kind matches the kind of formal argument κ_2, producing a constructor of the result kind κ_2.

The rules for type constructor equivalence create a congruent equivalence relation augmented with the β and η rules.

The formation and equivalence rules for types (in Table 2.3) are similar to those for constructors, except that as there are no type functions, no kinding is required. Only constructors of kind $\star$ may be converted to types with the injection $T()$. How this conversion happens is defined by the type equivalence rules. For example, the rule *(tyeq-int)* states that the type $T(\mathit{int})$ is equal to the type *int*. Again for readability in the examples, I will often omit the injection $T()$ when its use is apparent from context. For example, instead of $T(\alpha)\rightarrow \mathit{int}$, sometimes I will write the type as $\alpha \rightarrow \mathit{int}$.

Table 2.3: Core language: Static semantics

$\boxed{\Delta \vdash c : \kappa}$ **Constructor Formation**

[*c-var*] $$\frac{}{\Delta \vdash \alpha : \kappa} \; (\Delta(\alpha) = \kappa)$$

[*c-fn*] $$\frac{\Delta, \alpha{:}\kappa_1 \vdash c : \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.c : \kappa_1 \rightarrow \kappa_2} \; (\alpha \notin \mathit{Dom}(\Delta))$$

[*c-app*] $$\frac{\Delta \vdash c_1 : \kappa_1 \rightarrow \kappa_2 \qquad \Delta \vdash c_2 : \kappa_1}{\Delta \vdash c_1 c_2 : \kappa_2}$$

[*c-int-type*] $$\frac{}{\Delta \vdash \mathit{int} : \star}$$

[*c-arr-type*] $$\frac{}{\Delta \vdash \rightarrow : \star \rightarrow \star \rightarrow \star}$$

[*c-prod-type*] $$\frac{}{\Delta \vdash \times : \star \rightarrow \star \rightarrow \star}$$

$\boxed{\Delta \vdash c_1 = c_2 : \kappa}$ **Constructor Equivalence**

[*ceq-β*] $$\frac{\Delta, \alpha{:}\kappa' \vdash c_1 : \kappa \quad \Delta \vdash c_2 : \kappa'}{\Delta \vdash (\lambda\alpha{:}\kappa'.c_1)c_2 = c_1[c_2/\alpha] : \kappa} \; (\alpha \notin \mathit{Dom}(\Delta))$$

Table 2.3 (Continued)

[*ceq-η*]	$$\frac{\Delta \vdash c : \kappa_1 \to \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.c\,\alpha = c : \kappa_1 \to \kappa_2}\ (\alpha \notin Dom(\Delta))$$
[*ceq-cong1*]	$$\frac{\Delta, \alpha{:}\kappa \vdash c = c' : \kappa'}{\Delta \vdash \lambda\alpha{:}\kappa.c = \lambda\alpha{:}\kappa.c' : \kappa \to \kappa'}$$
[*ceq-cong2*]	$$\frac{\Delta \vdash c_1 = c_1' : \kappa' \to \kappa \qquad \Delta \vdash c_2 = c_2' : \kappa'}{\Delta \vdash c_1 c_2 = c_1' c_2' : \kappa}$$
[*ceq-ref*]	$$\frac{}{\Delta \vdash c = c : \kappa}$$
[*ceq-sym*]	$$\frac{\Delta \vdash c' = c : \kappa}{\Delta \vdash c = c' : \kappa}$$
[*ceq-trans*]	$$\frac{\Delta \vdash c = c' : \kappa \qquad \Delta \vdash c' = c'' : \kappa}{\Delta \vdash c = c'' : \kappa}$$
$\boxed{\Delta \vdash \sigma}$	**Type Formation**
[*t-con*]	$$\frac{\Delta \vdash c : \star}{\Delta \vdash T(c)}$$
[*t-int*]	$$\frac{}{\Delta \vdash int}$$
[*t-arrow*]	$$\frac{\Delta \vdash \sigma_1 \qquad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \to \sigma_2}$$
[*t-prod*]	$$\frac{\Delta \vdash \sigma_1 \qquad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \times \sigma_2}$$
[*t-all*]	$$\frac{\Delta, \alpha{:}\kappa \vdash \sigma}{\Delta \vdash \forall\alpha{:}\kappa.\sigma}\ (\alpha \notin Dom(\Delta))$$

Table 2.3 (Continued)

[*t-ex*]	$\dfrac{\Delta, \alpha{:}\kappa \vdash \sigma}{\Delta \vdash \exists\alpha{:}\kappa.\sigma}\ (\alpha \notin Dom(\Delta))$
$\boxed{\Delta \vdash \sigma_1 = \sigma_2}$	**Type Equivalence**
[*te-con*]	$\dfrac{\Delta \vdash c_1 = c_2 : \kappa}{\Delta \vdash T(c_1) = T(c_2)}$
[*te-int*]	$\dfrac{}{\Delta \vdash T(int) = int}$
[*te-arrow*]	$\dfrac{}{\Delta \vdash T(\rightarrow c_1 c_2) = T(c_1) \rightarrow T(c_2)}$
[*te-prod*]	$\dfrac{}{\Delta \vdash T(\times c_1 c_2) = T(c_1) \times T(c_2)}$
[*te-cong1*]	$\dfrac{\Delta \vdash \sigma_1 = \sigma_1' \qquad \Delta \vdash \sigma_2 = \sigma_2'}{\Delta \vdash \sigma_1 \rightarrow \sigma_2 = \sigma_1' \rightarrow \sigma_2'}$
[*te-cong2*]	$\dfrac{\Delta \vdash \sigma_1 = \sigma_1' \qquad \Delta \vdash \sigma_2 = \sigma_2'}{\Delta \vdash \sigma_1 \times \sigma_2 = \sigma_1' \times \sigma_2'}$
[*te-cong3*]	$\dfrac{\Delta, \alpha{:}\kappa \vdash \sigma = \sigma'}{\Delta \vdash \forall\alpha{:}\kappa.\sigma = \forall\alpha{:}\kappa.\sigma'}$
[*te-cong4*]	$\dfrac{\Delta, \alpha{:}\kappa \vdash \sigma = \sigma'}{\Delta \vdash \exists\alpha{:}\kappa.\sigma = \exists\alpha{:}\kappa.\sigma'}$
[*te-refl*]	$\dfrac{}{\Delta \vdash \sigma = \sigma}$
[*te-sym*]	$\dfrac{\Delta \vdash \sigma' = \sigma}{\Delta \vdash \sigma = \sigma'}$

Table 2.3 (Continued)

[*te-trans*]	$$\frac{\Delta \vdash \sigma = \sigma' \qquad \Delta \vdash \sigma' = \sigma''}{\Delta \vdash \sigma = \sigma''}$$
$\boxed{\Delta; \Gamma \vdash e : \sigma}$	**Term Formation**
[*e-int*]	$$\frac{}{\Delta; \Gamma \vdash i : int}$$
[*e-var*]	$$\frac{}{\Delta; \Gamma \vdash x : \sigma}\ (\Gamma(x) = \sigma)$$
[*e-fn*]	$$\frac{\Delta; \Gamma, x{:}\sigma_2 \vdash e : \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta; \Gamma \vdash \lambda x{:}\sigma_2.e : \sigma_2 \to \sigma_1}\ (x \notin Dom(\Gamma))$$
[*e-app*]	$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_2 \to \sigma_1 \quad \Delta; \Gamma \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash e_1 e_2 : \sigma_1}$$
[*e-fix*]	$$\frac{\Delta; \Gamma, f{:}\sigma \vdash e : \sigma \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash \mathit{fix}\ f{:}\sigma.\, e : \sigma}\ \left(\ \sigma \equiv \forall \alpha{:}\kappa.\sigma' \text{ or } \sigma \equiv \sigma_1 \to \sigma_2\ \right)$$
[*e-pair*]	$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \quad \Delta; \Gamma \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2}$$
[*e-pi1*]	$$\frac{\Delta; \Gamma \vdash e : \sigma_1 \times \sigma_2}{\Delta; \Gamma \vdash \pi_1 e : \sigma_1}$$
[*e-pi2*]	$$\frac{\Delta; \Gamma \vdash e : \sigma_1 \times \sigma_2}{\Delta; \Gamma \vdash \pi_2 e : \sigma_2}$$
[*e-tapp*]	$$\frac{\Delta; \Gamma \vdash e : \forall \alpha{:}\kappa.\sigma \quad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash e[c] : \sigma[c/\alpha]}$$
[*e-tfn*]	$$\frac{\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \Lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.\sigma}\ (\alpha \notin Dom(\Delta))$$

Table 2.3 (Continued)

[*e-pack*]	$$\frac{\Delta, \alpha{:}\kappa \vdash \sigma \qquad \Delta \vdash c : \kappa \qquad \Delta; \Gamma \vdash e : \sigma[c/\alpha]}{\Delta; \Gamma \vdash \mathit{pack}\langle c, e\rangle\ \mathit{as}\ \exists\alpha{:}\kappa.\sigma : \exists\alpha{:}\kappa.\sigma}\ (\alpha \notin \mathit{Dom}(\Delta))$$
[*e-unpack*]	$$\frac{\Delta; \Gamma \vdash e_1 : \exists\alpha{:}\kappa.\sigma_2 \qquad \Delta, \alpha{:}\kappa; \Gamma, x{:}\sigma_2 \vdash e_2 : \sigma_1}{\Delta; \Gamma \vdash \mathit{unpack}\langle \alpha, x\rangle = e_1\ \mathit{in}\ e_2 : \sigma_1}\ (\alpha, x \notin \mathit{Dom}(\Delta; \Gamma))$$
[*e-equiv*]	$$\frac{\Delta; \Gamma \vdash e : \sigma_2 \quad \Delta \vdash \sigma_1 = \sigma_2}{\Delta; \Gamma \vdash e : \sigma_1}$$

Finally, the terms of LI are similar to those of other typed lambda calculi in formation (see Table 2.3) and behavior (see Table 2.4). Like the type constructor language, the term level also includes functions $\lambda x{:}\sigma.e$ of function type $(\sigma \rightarrow \sigma')$. Additionally, the term level includes integer constants, pairs, two forms of type abstraction and a way to define recursive functions, listed in detail below.

Constants The integers $1, 2, 3, \ldots$ (represented by the metavariable i) are all of type *int*.

Products Product types $\sigma_1 \times \sigma_2$ are created by pairing an expression e_1 of type σ_1 with an expression e_2 of type σ_2. The first and second components of a pair are projected with π_1 and π_2 respectively.

Universal types Terms may abstract type constructors of any kind, with $\Lambda\alpha{:}\kappa.e$. The type variable α is bound within e. This form explicitly supports polymorphism. The type variable α may be instantiated with any type. During execution, a type application $e[c]$, substitutes the type argument c for the bound variable.

Existential types Terms may also hide the identity of a type constructor within an *existential package*, of type $\exists\alpha{:}\kappa.\sigma$. This form is the dual to universal types above. If a term e has type $\sigma[c/\alpha]$, then the term *pack* $\langle c, e\rangle$ *as* $\exists\alpha{:}\kappa.\sigma$, creates such an existential package, hiding the type constructor c. Terms of existential type must be opened before they are used, though the hidden type remains abstract. The term *let* $\langle \alpha, y\rangle = e_1$ *in* e_2 opens the package e_1

inside the term e_2, binding the constructor variable α to the hidden type c and the term variable y to the packed term e. Without type analysis, the identity of the type constructor α must remain unknown [MP88]. However, in languages (such as LI) that support type analysis, this hidden type may be determined.

Recursion The fixed points of recursive terms are created with the term *fix*. This term is well formed if the type of the bound variable (the fixed point) is the same as the type of the entire expression. This type must either be a polymorphic or function type.

2.2.3 Operational semantics

Table 2.4: Core language: Operational semantics

[*ev-β*]	$(\lambda x{:}\sigma.e)v \mapsto e[v/x]$
[*ev-ty-β*]	$(\Lambda\alpha{:}\kappa.e)[c] \mapsto e[c/\alpha]$
[*ev-tapp*]	$(\mathit{fix}\ f{:}\sigma.e)v \mapsto (e[\mathit{fix}\ f{:}\sigma.e/f])v$
[*ev-fix*]	$(\mathit{fix}\ f{:}\sigma.e)[c] \mapsto (e[\mathit{fix}\ f{:}\sigma.e/f])[c]$
[*ev-pi1*]	$\pi_1\langle v_1, v_2\rangle \mapsto v_1$
[*ev-pi2*]	$\pi_2\langle v_1, v_2\rangle \mapsto v_2$
[*ev-unpack*]	$\mathit{unpack}\langle\alpha, x\rangle = (\mathit{pack}\ v\ \mathit{as}\ \exists\beta.\sigma_1\ \mathit{hiding}\ \sigma_2)\ \mathit{in}\ e_2 \mapsto e_2[\sigma_2/\alpha, v/x]$
[*ev-app1*]	$\dfrac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}$

Table 2.4 (Continued)

[*ev-app2*]	$\dfrac{e \mapsto e'}{ve \mapsto ve'}$
[*ev-tapp*]	$\dfrac{e \mapsto e'}{e[c] \mapsto e'[c]}$
[*ev-pi*]	$\dfrac{e \mapsto e'}{\pi_i e \mapsto \pi_i e'}$
[*ev-pair1*]	$\dfrac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}$
[*ev-pair2*]	$\dfrac{e \mapsto e'}{\langle v, e\rangle \mapsto \langle v, e'\rangle}$
[*ev-pack*]	$\dfrac{e \mapsto e'}{pack\langle c, e\rangle\ as\ \exists\beta.\sigma \mapsto pack\langle c, e'\rangle\ as\ \exists\beta.\sigma}$
[*ev-unpack2*]	$\dfrac{e \mapsto e'}{unpack\langle\alpha, x\rangle = e\ in\ e_2 \mapsto unpack\langle\alpha, x\rangle = e'\ in\ e_2}$

Harper and Morrisett designed LI to be an intermediate language for a high-performance ML compiler. Therefore, they formalized its computational behavior with a small-step, call-by-value operational semantics. This semantics defines the evaluation of an expression with the transition relation $e \mapsto e'$. (This notation will be the evaluation relation for all languages of this dissertation. I will use $\mapsto_i$ to refer only to LI evaluation, when it is not clear from context.) This relation states that the term e evaluates in one step to the term e'. The transitive closure of this relation (notated $\mapsto^*$) describes the execution sequences of the term language.

The choice of call-by-value is not an important decision in the design of type-analyzing languages. Lazy versions of LI may also be defined. However, as LI evaluation is call-by-value (also known as eager evaluation), the arguments to LI functions must be fully evaluated before the functions are applied to them (refer to rule *ev-β* in Table 2.4). The closed forms of LI (those that are well formed in the empty context) that furthermore do not step to any other terms are called *values*.

Table 2.5: LI: Operational semantics of *typerec*

$$[\textit{ev-trec-int}] \quad \frac{c \text{ normalizes to } \textit{int}}{\textit{typerec}[\alpha.\sigma]\ c\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times}) \mapsto_i e_{\textit{int}}}$$

$$[\textit{ev-trec-arrow}] \quad \frac{c \text{ normalizes to } (c_1 \rightarrow c_2)}{\begin{array}{l}\textit{typerec}[\alpha.\sigma]\ c\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times}) \mapsto_i \\ \quad e_{\rightarrow}[c_1]\ (\textit{typerec}[\alpha.\sigma]\ c_1\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times})) \\ \quad\quad [c_2]\ (\textit{typerec}[\alpha.\sigma]\ c_2\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times}))\end{array}}$$

$$[\textit{ev-trec-prod}] \quad \frac{c \text{ normalizes to } (c_1 \times c_2)}{\begin{array}{l}\textit{typerec}[\alpha.\sigma]\ c\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times}) \mapsto_i \\ \quad e_{\times}[c_1]\ (\textit{typerec}[\alpha.\sigma]\ c_1\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times})) \\ \quad\quad [c_2]\ (\textit{typerec}[\alpha.\sigma]\ c_2\ (e_{\textit{int}}, e_{\rightarrow}, e_{\times}))\end{array}}$$

They may be described by the abstract syntax in Table 2.2.1. By examination of the evaluation rules, we see that none of these terms steps to any others. Later I will discuss a proof that this syntax describes all of the closed terms for which no reduction rules apply. Additionally, products in LI are also treated eagerly, both components must be fully evaluated before projection. Following Harper [Har01], $\textit{fix}\, x : \sigma.e$ is a value that unfolds itself when applied to type or term arguments. Therefore, the type of the expression e must be a polymorphic or function type.

If e is closed and well typed, a series of these steps will either eventually diverge or produce a value. This property is the principle of type soundness, discussed in Section 2.5.

2.3 Type analysis operators

The important additions to LI are the type analysis operators that analyze type constructors of kind $\star$: *typerec* produces terms and *Typerec* produces other type constructors. Morrisett, in his dissertation [Mor95], describes these operations as folds over an inductively defined data-structure, the kind $\star$:

> 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 elimination forms for the kind $\star$ at the constructor and term level respectively. The introductory forms at the constructor level are the constructors of kind $\star$;

Table 2.6: LI: Schema for *typerec* branches

$$[\alpha.\sigma]\langle c : \star\rangle = \sigma[c/\alpha]$$
$$[\alpha.\sigma]\langle c : \kappa_1 \to \kappa_2\rangle = \forall\alpha{:}\kappa_1.[\alpha.\sigma]\langle \alpha : \kappa_1\rangle \to [\alpha.\sigma]\langle (c\alpha) : \kappa_2\rangle$$

> there are no introductory forms at the term level to preserve the phase distinction. In effect, *Typerec* and *typerec* let us *fold* some computation over a monotype. Limiting this 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.

We see the truth in this description in the operational semantics for *typerec* (Table 2.5) and in the rules describing the equational theory of *Typerec* (Table 2.8). These operators, when given an argument type constructor c, dispatch to one of their branches based on whether c normalizes to *int*, a function type or a product type. The definition of constructor normalization (the transitive closure of the constructor reduction relation $\leadsto$ in Table 2.9) is based on the equality relation and is described in more detail below.

Furthermore, *typerec* and *Typerec* iteratively continue through the subcomponents of the argument type constructor. For example, the term (where the notation $[\alpha.\sigma]$ is for type checking and $\overline{e}$ abbreviates $(e_{int}, e_{\to}, e_{\times})$)

$$\mathit{typerec}[\alpha.\sigma]\ (\mathit{int}\ \to\ (\mathit{int}\ \times\ \mathit{int}))\ \overline{e}$$

steps to

$$e_{\to}\ [\mathit{int}]\ (\mathit{typerec}[\alpha.\sigma]\ \ \mathit{int}\ \ \overline{e})\ [\mathit{int}\ \times\ \mathit{int}]\ (\mathit{typerec}[\alpha.\sigma]\ (\mathit{int}\ \times\ \mathit{int})\ \overline{e})$$

Above, the arrow branch is applied to the first constructor argument, the term argument that represent iteration over that constructor, the second constructor argument and the term argument that represent iteration over that constructor. This pattern of operation over an inductive datatype is known as a paramorphism [Mee92] or primitive recursion: each branch receives the subcomponents of the type as well as the continuation of iteration.

When are *typerec* terms well formed? The annotation $[\alpha.\sigma]$ permits syntax-directed type-checking of *typerec* terms. This annotation expresses the relationship

Table 2.7: LI: Static semantics of *typerec*

$$[e\text{-}trec]\quad \frac{\begin{array}{c}\Delta \vdash c : \star \\ \Delta, \alpha{:}\star \vdash \sigma \\ \Delta; \Gamma \vdash e_{int} : [\alpha.\sigma]\langle int : \star\rangle \\ \Delta; \Gamma \vdash e_{\rightarrow} : [\alpha.\sigma]\langle \rightarrow : \star \rightarrow \star \rightarrow \star\rangle \\ \Delta; \Gamma \vdash e_{\times} : [\alpha.\sigma]\langle \times : \star \rightarrow \star \rightarrow \star\rangle\end{array}}{\Delta; \Gamma \vdash typerec[\alpha.\sigma]\, c\, (e_{int}, e_{\rightarrow}, e_{\times}) : \sigma[c/\alpha]}$$

between the analyzed type and the return type of the term and allows us to describe the appropriate types of the branches of the *typerec*. We use the schema $[\alpha.\sigma]\langle c : \kappa\rangle$ to represent the result of a branch on constructor c of kind κ. This schema is defined in Table 2.6 by induction on κ.

Using this schema, we represent the branch type indexed by the kind of the constructor c. If that kind is $\star$, then we just substitute c for α in σ. For example, the type of the e_{int} branch is $\sigma[int/\alpha]$, which reflects that in that branch we may assume the argument constructor is *int*. If the constructor is of a higher kind, it is parameterized by other types, and so the branch for that constructor must use the result of the induction on the subcomponents in computing the branch for that type. For example, the type of the branch $e_{\rightarrow}$ is $[\alpha.\sigma]\langle \rightarrow : \star \rightarrow \star \rightarrow \star\rangle$ which is equivalent to $\forall\beta{:}\star.\sigma[\beta/\alpha] \rightarrow \forall\gamma{:}\star.\sigma[\gamma/\alpha] \rightarrow \sigma[(\beta \rightarrow \gamma)/\alpha]$. Because the types of the branches are represented with this schema, it is easy to extend *typerec* with branches for other type constructor constants (such as *unit*, *bool*, or $+$).

Table 2.8: LI: Static semantics of *Typerec*

$\boxed{\Delta \vdash c : \kappa}$

$$[c\text{-}trec]\quad \frac{\begin{array}{c}\Delta \vdash c : \star \\ \Delta \vdash c_{int} : \kappa\langle\star\rangle \\ \Delta \vdash c_{\rightarrow} : \kappa\langle \star \rightarrow \star \rightarrow \star\rangle \\ \Delta \vdash c_{\times} : \kappa\langle \star \rightarrow \star \rightarrow \star\rangle\end{array}}{\Delta \vdash Typerec\ c\, (c_{int}, c_{\rightarrow}, c_{\times}) : \kappa}$$

Table 2.8 (Continued)

$\boxed{\Delta \vdash c = c' : \kappa}$

[*ceq-trec-int*]

$$\frac{\Delta \vdash \mathit{Typerec}\ \mathit{int}\ (c_{int}, c_{\rightarrow}, c_{\times}) : \kappa}{\Delta \vdash \mathit{Typerec}\ \mathit{int}\ (c_{int}, c_{\rightarrow}, c_{\times}) = c_{int} : \kappa}$$

[*ceq-trec-arrow*]

$$\frac{\Delta \vdash \mathit{Typerec}\ (\rightarrow c_1 c_2)\ (c_{int}, c_{\rightarrow}, c_{\times}) : \kappa}{\begin{array}{l}\Delta \vdash \mathit{Typerec}\ (\rightarrow c_1 c_2)\ (c_{int}, c_{\rightarrow}, c_{\times}) = \\ \quad c_{\rightarrow}\ \ c_1\ (\mathit{Typerec}\ \ c_1\ (c_{int}, c_{\rightarrow}, c_{\times})) \\ \quad\quad c_2\ (\mathit{Typerec}\ \ c_2\ (c_{int}, c_{\rightarrow}, c_{\times})) : \kappa\end{array}}$$

[*ceq-trec-prod*]

$$\frac{\Delta \vdash \mathit{Typerec}\ (\times c_1 c_2)\ (c_{int}, c_{\rightarrow}, c_{\times}) : \kappa}{\begin{array}{l}\Delta \vdash \mathit{Typerec}\ (\times c_1 c_2)\ (c_{int}, c_{\rightarrow}, c_{\times}) = \\ \quad c_{\times}\ \ c_1\ (\mathit{Typerec}\ \ c_1\ (c_{int}, c_{\rightarrow}, c_{\times})) \\ \quad\quad c_2\ (\mathit{Typerec}\ \ c_2\ (c_{int}, c_{\rightarrow}, c_{\times})) : \kappa\end{array}}$$

[*ceq-trec-cong*]

$$\frac{\begin{array}{c}\Delta \vdash c = c' : \star \\ \Delta \vdash c_{int} = c'_{int} : \kappa\langle\star\rangle \\ \Delta \vdash c_{\rightarrow} = c'_{\rightarrow} : \kappa\langle\star \rightarrow \star \rightarrow \star\rangle \\ \Delta \vdash c_{\times} = c'_{\times} : \kappa\langle\star \rightarrow \star \rightarrow \star\rangle\end{array}}{\Delta \vdash \mathit{Typerec}\ \ c\ (c_{int}, c_{\rightarrow}, c_{\times}) = \mathit{Typerec}\ \ c'\ (c'_{int}, c'_{\rightarrow}, c'_{\times}) : \kappa}$$

Table 2.9: LI: Constructor reduction

$\boxed{c_1 \leadsto c_2}$

[*cn-β*]

$$\frac{}{(\lambda\alpha{:}\kappa'.c_1)c_2 \leadsto c_1[c_2/\alpha]}$$

[*cn-η*]

$$\frac{}{\lambda\alpha{:}\kappa_1.c\,\alpha \leadsto c}$$

[*cn-cong1*]

$$\frac{c \leadsto c'}{\lambda\alpha{:}\kappa.c \leadsto \lambda\alpha{:}\kappa.c'}$$

[*cn-cong2*]

$$\frac{c_1 \leadsto c_1' \qquad c_2 \leadsto c_2'}{c_1 c_2 \leadsto c_1' c_2'}$$

[*cn-trec-int*]

$$\frac{}{\mathit{Typerec}\ \mathit{int}\ (c_{\mathit{int}}, c_{\to}, c_{\times}) \leadsto c_{\mathit{int}}}$$

[*cn-trec-arrow*]

$$\frac{}{\begin{array}{l}\mathit{Typerec}\,(\to c_1 c_2)\,(c_{\mathit{int}}, c_{\to}, c_{\times}) \leadsto \\ \quad c_{\to}\ c_1\,(\mathit{Typerec}\,c_1\,(c_{\mathit{int}}, c_{\to}, c_{\times})) \\ \quad\ c_2\ (\mathit{Typerec}\,c_2\,(c_{\mathit{int}}, c_{\to}, c_{\times}))\end{array}}$$

[*cn-trec-prod*]

$$\frac{}{\begin{array}{l}\mathit{Typerec}\,(\times c_1 c_2)\,(c_{\mathit{int}}, c_{\to}, c_{\times}) \leadsto \\ \quad c_{\times}\ c_1\,(\mathit{Typerec}\,c_1\,(c_{\mathit{int}}, c_{\to}, c_{\times})) \\ \quad\ c_2\ (\mathit{Typerec}\,c_2\,(c_{\mathit{int}}, c_{\to}, c_{\times}))\end{array}}$$

[*cn-trec-cong*]

$$\frac{c \leadsto c' \qquad c_{\mathit{int}} \leadsto c_{\mathit{int}}' \qquad c_{\to} \leadsto c_{\to}' \qquad c_{\times} \leadsto c_{\times}'}{\mathit{Typerec}\ \ c\ (c_{\mathit{int}}, c_{\to}, c_{\times}) \leadsto \mathit{Typerec}\ \ c'\ (c_{\mathit{int}}', c_{\to}', c_{\times}')}$$

The kinding rule for *Typerec* is natural. To compute a constructor of kind κ, present a type argument and three branches that when fully applied return κ constructors. Again, the kinding of each branch depends on the kind of the constructor matched. We use the notation $\kappa\langle\kappa'\rangle$ to describe the kind of a branch matching a constructor of kind κ' in a *Typerec* expression producing kind κ. Because *Typerec* computes a paramorphic fold over its argument, the branches for constructors of higher kinds require both the subcomponent of the constructor and the result of analysis of the subcomponent of the constructor. For example, for the arrow branch, $\kappa\langle\star \to \star \to \star\rangle$ is equivalent to $\star \to \kappa \to \star \to \kappa \to \kappa$.

$$\begin{aligned}
&\kappa\langle\star\rangle = \kappa \\
&\kappa\langle\kappa_1 \to \kappa_2\rangle = \kappa_1 \to \kappa\langle\kappa_1\rangle \to \kappa\langle\kappa_2\rangle
\end{aligned}$$

The equivalence rules for *Typerec* are similar to the operational semantics of *typerec*. The equivalence of constructors also derives the constructor reduction relation in Table 2.9. Each constructor that matches the expression on the left hand side of the rule, is rewritten to the constructor that matches the right hand side. For example, the constructor $\mathit{Typerec}\ \mathit{int}\ (c_{int}, c_{\to}, c_{\times})$ reduces to c_{int}.

2.4 Formalizing the examples

Now that we have fully defined the LI language, we may formalize the examples of the first section. However, for each example, we need new forms of types and terms that we have not included in the core language. For example, in order to implement polymorphic equality in LI, we need to add Boolean values, logical operations over Booleans, a primitive equality function for integers and a void type.

With these additions, we may formalize polymorphic equality example as follows.

$$\begin{aligned}
&\mathit{Eq} \stackrel{\text{def}}{=} \lambda\alpha{:}\star\,.\,\mathit{Typerec}\,\alpha\ \mathit{of} \\
&\quad \mathit{int} \quad \Rightarrow \mathit{int} \\
&\quad \to \quad \Rightarrow \lambda\beta_1{:}\star\,.\lambda\beta_2{:}\star\,.\lambda\gamma_1{:}\star\,.\lambda\gamma_2{:}\star\,.\,\mathit{void} \\
&\quad \times \quad \Rightarrow \lambda\beta_1{:}\star\,.\lambda\beta_2{:}\star\,.\lambda\gamma_1{:}\star\,.\lambda\gamma_2{:}\star\,.\beta_2 \times \gamma_2 \\
\\
&\mathit{eq} : \forall\alpha{:}\star\,.\,\mathit{Eq}\,\alpha \to \mathit{Eq}\,\alpha \to \mathit{bool} \\
&\mathit{eq} \stackrel{\text{def}}{=} \Lambda\alpha{:}\star\,.\,\mathit{typerec}[\alpha.\,\mathit{Eq}\,\alpha \to \mathit{Eq}\,\alpha \to \mathit{bool}]\ \alpha\ \mathit{of} \\
&\quad \mathit{int} \quad \Rightarrow \mathit{primIntEq} \\
&\quad \to \quad \Rightarrow \Lambda\beta{:}\star\,.\lambda r_\beta.\Lambda\gamma{:}\star\,.\lambda r_\gamma. \\
&\qquad\qquad \lambda x{:}\,\mathit{void}\,.\lambda y{:}\,\mathit{void}\,.\,\mathit{true} \\
&\quad \times \quad \Rightarrow \Lambda\beta{:}\star\,.\lambda r_\beta{:}\,\mathit{Eq}\,\beta \to \mathit{Eq}\,\beta \to \mathit{bool}\,. \\
&\qquad\qquad \Lambda\gamma{:}\star\,.\lambda r_\gamma{:}\,\mathit{Eq}\,\gamma \to \mathit{Eq}\,\gamma \to \mathit{bool}\,. \\
&\qquad\qquad\quad \lambda x{:}\,\mathit{Eq}(\beta \times \gamma).\lambda y{:}\,\mathit{Eq}(\beta \times \gamma). \\
&\qquad\qquad\qquad r_\beta(\pi_1 x)(\pi_1 y)\ \&\&\ r_\gamma(\pi_2 x)(\pi_2 y)
\end{aligned}$$

For the other examples in the beginning of this chapter, other forms of types and terms are also necessary. I have omitted these forms from the formal language not because they are difficult to model, but because the semantics of these terms are closely related to that of the terms previously described. Their addition does not change the properties of LI (or any of the subsequent languages of this dissertation) in relation to type analysis. In these examples, the *typerec* and *Typerec* terms may also be extended with new branches for the new type forms.

Because these additional forms will also be necessary for future examples, I describe them in more detail below.

Void The type *void* contains no values. Any expression of this type must diverge.

Unit The type *unit* contains the single value ().

Bool The type *bool* contains two values, *true* and *false*.

String The type *string* contains string constants such as `"hello"`, `"peripatetic"` and the empty string `""`. Terms that operate over strings include string concatenation ($++$) and string equality (`==`).

Arrays Arrays are of type $array\,\alpha$ where α is the type of the array elements. They may be accessed with the subscript operator $sub : array\,\alpha \times int \rightarrow \alpha$ and updated with the operator $set : array\,\alpha \times int \times \alpha \rightarrow unit$.

Sums (co-products) Disjoint sums, $\sigma_1 + \sigma_2$ are created by using either the first or second injection (inj_1 and inj_2) with a term of type σ_1 or of type σ_2 respectively. They are eliminated by case analysis. If e is of type $\sigma_1 + \sigma_2$, e_1 of type $\sigma_1 \rightarrow \sigma_3$, and e_2 of type $\sigma_2 \rightarrow \sigma_3$ then the term $case\ e\ e_1\ e_2$ is of type σ_3. However, to enhance the readability of the examples, I will also use the pattern matching syntax of ML [MTHM97] for the creation and elimination of sums.

Static semantics

$$[\textit{e-inj1}] \quad \frac{\Delta; \Gamma \vdash e : \sigma_1 \qquad \Delta \vdash \sigma_2}{\Delta; \Gamma \vdash inj_1^{\sigma_1+\sigma_2}\, e : \sigma_1 + \sigma_2}$$

$$[\textit{e-inj2}] \quad \frac{\Delta; \Gamma \vdash e : \sigma_2 \qquad \Delta \vdash \sigma_1}{\Delta; \Gamma \vdash inj_2^{\sigma_1+\sigma_2}\, e : \sigma_1 + \sigma_2}$$

$$[\textit{e-case}] \quad \frac{\Delta; \Gamma \vdash e : \sigma_1 + \sigma_2 \qquad \Delta; \Gamma \vdash e_1 : \sigma_1 \rightarrow \sigma \qquad \Delta; \Gamma \vdash e_2 : \sigma_2 \rightarrow \sigma}{\Delta; \Gamma \vdash case\ e\ e_1\ e_2 : \sigma}$$

Dynamic semantics

$$[\textit{ev-inj}_1] \quad \textit{case}(\textit{inj}_1^{\sigma_1+\sigma_2}\ v)\ e_1\ e_2 \mapsto e_1 v$$

$$[\textit{ev-inj}_2] \quad \textit{case}(\textit{inj}_2^{\sigma_1+\sigma_2}\ v)\ e_1\ e_2 \mapsto e_2\ v$$

$$[\textit{ev-cong-inj1}] \quad \frac{e \mapsto e'}{\textit{inj}_1^{\sigma_1+\sigma_2}\ e \mapsto \textit{inj}_1^{\sigma_1+\sigma_2}\ e'}$$

$$[\textit{ev-cong-inj2}] \quad \frac{e \mapsto e'}{\textit{inj}_2^{\sigma_1+\sigma_2}\ e \mapsto \textit{inj}_2^{\sigma_1+\sigma_2}\ e'}$$

$$[\textit{ev-cong-case}] \quad \frac{e \mapsto e'}{\textit{case}\ e\ e_1\ e_2 \mapsto \textit{case}\ e'\ e_1\ e_2}$$

Recursive types Parameterized recursive types are written $\mu_k(c_1, c_2)$, where k is the parameter kind and c_1 is a type constructor with kind $(k \to \star) \to (k \to \star)$. Intuitively, c_1 recursively defines a type constructor with kind $k \to \star$, which is then instantiated with the parameter c_2 (having kind k). Thus, members of $\mu_k(c_1, c_2)$ unfold into the type $c_1(\lambda\alpha{:}\kappa.\mu_k(c_1, \alpha))c_2$ and fold the opposite way. The special case of non-parameterized recursive types are defined as $\mu\alpha.\sigma = \mu_1(\lambda\varphi{:}1 \to \star.\ \lambda\beta{:}1.\ \sigma[\varphi(*)/\alpha], *)$.

Because recursive types bind a type variable, they cannot be described by an inductive type constructor as the other types can. Therefore, in LI, they must only be added to the type language, and not represented by the type constructor language. In Chapters 4 and 6, I will go into detail about the inclusion of such quantified types in the analyzable part of the language.

Static semantics

$$[\textit{e-unroll}] \quad \frac{\Delta; \Gamma \vdash e : \mu_k(c, c')}{\Delta; \Gamma \vdash \textit{unroll}\ e : c(\lambda\alpha{:}k.\mu_k(c, \alpha))c'}$$

$$[\textit{e-roll}] \quad \frac{\Delta; \Gamma \vdash e : c(\lambda\alpha{:}k.\mu_k(c, \alpha))c' \qquad \Delta \vdash \mu_k(c, c') : \star}{\Delta; \Gamma \vdash \textit{roll}_{\mu_k(c,c')}\ e : \mu_k(c, c')}$$

Dynamic semantics

$$[\textit{ev-roll-}\beta] \quad \textit{unroll}(\textit{roll}_{\mu_k(c,c')}\, v) \mapsto v$$

$$[\textit{ev-cong-roll}] \quad \frac{e \mapsto e'}{\textit{roll}_{\mu_k(c,c')}\, e \mapsto \textit{roll}_{\mu_k(c,c')}\, e'}$$

$$[\textit{ev-cong-unroll}] \quad \frac{e \mapsto e'}{\textit{unroll}\, e \mapsto \textit{unroll}\, e'}$$

2.5 Typing properties of LI

Morrisett [Mor95] proves several theorems about the properties of well-formed terms of the LI language. The two most important are that type checking LI terms is decidable and that the type system is sound with respect to the operational semantics. These properties are essential for any typed calculus. The first means that for any expression we can effectively tell whether there exists a derivation that the expression is well formed. The second, known as *type soundness* means that as the program executes, type errors will not occur.

The changes that I have made to the language in this chapter (mostly dealing with the operational semantics and the addition of a few new term forms) do not invalidate those theorems. Furthermore, these properties are also true of the languages I will describe in later chapters. In those chapters, I will prove that these properties hold with the same techniques that Morrisett employed for LI. Therefore, as an introduction to these proof techniques, I will give a quick overview of proofs of those theorems and the key lemmas that support them.

2.5.1 Decidable type checking

The proof of the decidability of type checking in LI is complicated by the term formation rule (*e-equiv*) that allows the replacement of a term's type with any other equivalent type at any point in the derivation. All other term formation rules require the derivation of well formedness for some subterm of the conclusion, therefore, the syntax of the term guides the search for the derivation. However, this equivalence rule requires the derivation of well formedness of the *same term* at a new type. Therefore, type checking is not syntax directed.

To solve this dilemma, Morrisett proved that every type has a unique normal form. Then, he defines the notion of a *normal derivation*. A derivation is normal if at every step the type of the right side of the ":" is replaced with its normal form. Deciding if a term has a normal derivation is entirely syntax directed, because we know where to employ *e-equiv.* Furthermore, every term is well formed (has a derivation ascribing some type), if and only if it has a normal derivation. Therefore, type-checking reduces to verifying the existence of a normal derivation.

A large part of this is determining the normal form for type constructors. Following standard techniques [Gan86, Tai67], Morrisett develops a set of reduction rules (Table 2.9) that may be used to convert a constructor to its normal form. Using these rules, Morrisett proved the following properties about LI type constructors:

Theorem 2.5.1 (Morrisett) *1. Every constructor has a unique normal form.*

2. If a constructor is well formed, there is an algorithm to calculate its normal form.

3. Equivalence of well-formed constructors is decidable.

Normal forms for LI types are a direct extension for normal forms for LI constructors. The algorithm to produce the normal form for a type is to normalize any constructor components and recursively replace $T(int)$ with int, $T(c_1 \rightarrow c_2)$ with $T(c_1) \rightarrow T(c_2)$ and $T(c_1 \times c_2)$ with $T(c_1) \times T(c_2)$.

Using the normal form for types and the algorithm described above Morrisett proves the theorem:

Theorem 2.5.2 (Decidability of LI type checking) *Given well-formed* $\Delta; \Gamma$ *and expression* e*, there is an algorithm to determine whether there exists a* σ *such that* $\Delta; \Gamma \vdash e : \sigma$ *is derivable in LI.*

2.5.2 Type soundness

Morrisett proves type soundness for LI syntactically, in the style of Wright and Felleisen [WF94]. This proof essentially shows that if a term type checks, then the operational semantics will not get *stuck.* A term is considered stuck if it is not a value and no rule of the operational semantics applies to it.

The proof of type soundness requires a number of technical lemmas. First, we must show that substitution does not destroy the well formedness of expressions.

Lemma 2.5.3 (Substitution) *1. If* $\Delta, \alpha{:}\kappa' \vdash c : \kappa$ *and* $\Delta \vdash c' : \kappa'$ *then* $\Delta[c'/\alpha] \vdash c[c'/\alpha] : \kappa$.

2. If $\Delta, \alpha{:}\kappa' \vdash c_1 = c_2 : \kappa$ *and* $\Delta \vdash c' : \kappa'$ *then* $\Delta[c'/\alpha] \vdash c_1[c'/\alpha] = c_2[c'/\alpha] : \kappa$.

3. If $\Delta, \alpha{:}\kappa \vdash \sigma$ *and* $\Delta \vdash c : \kappa$ *then* $\Delta[c/\alpha] \vdash \sigma[c/\alpha]$.

4. *If* $\Delta, \alpha{:}\kappa \vdash \sigma = \sigma'$ *and* $\Delta \vdash c : \kappa$ *then* $\Delta[c/\alpha] \vdash \sigma[c/\alpha] = \sigma'[c/\alpha]$.

5. *If* $\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$ *and* $\emptyset; \emptyset \vdash c : \kappa$ *then* $\Delta; \Gamma[c/\alpha] \vdash e[c/\alpha] : \sigma[c/\alpha]$.

6. *If* $\Delta; \Gamma, x{:}\sigma' \vdash e : \sigma$ *and* $\emptyset; \emptyset \vdash e' : \sigma'$ *then* $\Delta; \Gamma \vdash e[e'/x] : \sigma$.

Substitution

Proofs of these lemmas appear in Morrisett [Mor95]. □

These substitution lemmas are a large part of the proof of Subject Reduction Lemma below (also called Type Preservation). This lemma states that if a term is well formed and steps to another term in the operational semantics, the resulting term is also well formed at the same type.

Lemma 2.5.4 (Subject Reduction) *If* $\emptyset \vdash e : \sigma$ *and* $e \mapsto e'$ *then* $\emptyset \vdash e' : \sigma$.

Proof

Proof is by induction on the evaluation relation $e \mapsto e'$. □

The next lemma states that the forms of closed values can be determined by their types.

Lemma 2.5.5 (Canonical Forms) *If* $\emptyset \vdash v : \sigma$ *then*

1. *If* $\emptyset \vdash \sigma = int$ *then* v *is* i.

2. *If* $\emptyset \vdash \sigma = \sigma_1 \to \sigma_2$ *then* v *is either* $\lambda x{:}\sigma_1.e$ *or* $(\mathit{fix}\ f{:}(\sigma_1 \to \sigma_2).v')[c_1] \cdots [c_n]$.

3. *If* $\emptyset \vdash \sigma = \sigma_1 \times \sigma_2$ *then* v *is of the form* $\langle v_1, v_2 \rangle$.

4. *If* $\emptyset \vdash \sigma = \forall \alpha{:}\kappa.\sigma_1$ *then* v *is either* $\Lambda \alpha{:}\kappa.v'$ *or* $(\mathit{fix}\ f{:}(\alpha{:}\kappa.\sigma_1).v')[c_1] \cdots [c_n]$.

5. *If* $\emptyset \vdash \sigma = \exists \alpha{:}\kappa.\sigma_1$ *then* v *is* $\mathit{pack}\langle c, v' \rangle$ *as* $\exists \alpha.\sigma_1$.

Proof

Proof follows from examination of the normal derivations that produce values (in Morrisett [Mor95]). □

Because I have written the operational semantics differently than Morrisett, in order to prove Progress for the *typerec* rules presented here I must prove that all closed constructors are equivalent to a constructor of an appropriate form.

Lemma 2.5.6 (Closed Normal Forms) *If $\emptyset \vdash c : \star$ and c is in normal form then either*

1. $c = int$
2. $c = \rightarrow c_1 c_2$, *for some* c_1 *and* c_1
3. $c = \times c_1 c_2$, *for some* c_1 *and* c_1

Proof

Proof is by induction on the derivation $\emptyset \vdash c : \star$, noting that when c is not of one of these forms, then by induction, a reduction rule applies to c. □

The Progress Lemma below states that a well-typed term is either a value or able to take a step in the operational semantics.

Lemma 2.5.7 (Progress) *If $\emptyset \vdash e : \sigma$ and e is not a value then there exists an e' such that $e \mapsto e'$.*

Putting the above lemmas together, we may finally prove Type Soundness (also known as Type Safety) for LI.

Theorem 2.5.8 (Type Soundness) *If $\emptyset \vdash e : \sigma$ and $e \mapsto^* e'$ then e' is not stuck.*

Proof

By induction on n, the number of steps in the evaluation of $e \mapsto^* e'$. If n is zero, then Progress (2.5.7) states that e is not stuck. Otherwise, $e \mapsto e'' \mapsto^{n-1} e'$. By Subject Reduction (2.5.4) we can say that $\emptyset \vdash e'' : \sigma$ and then apply the inductive hypothesis to conclude that e' is not stuck. □

2.6 Discussion and chapter summary

The LI language provides a good basis for simply modeling run-time type analysis. With its type-passing operational semantics, the mechanisms necessary to support type dispatch are included in the single term *typerec*. Furthermore, the type constructor *Typerec* allows many uses of *typerec* to be assigned types. In a rich language, these operators can support a number of motivating examples of non-parametric programming. This chapter includes formalization of dynamic typing, type directed partial evaluation, polymorphic equality and flexible data representation within LI.

An additional purpose of this chapter is to provide background for the rest of the dissertation. This chapter describes a standard formalization of a core typed language. The terms of this language, including integers, products, functions, universally and existentially polymorphic terms, appear in many of languages of the subsequent chapters. It also briefly describes how to prove the two most important properties of this language. First, LI has decidable type checking. For any term, there is an effective procedure to determine if there is a derivation that it is well typed. Second, LI, is type sound. During execution of any well-typed LI term, it will either produce a value or run forever. These properties will also be shown for the subsequent languages.

Chapter 3

Type analysis without analyzing types

3.1 Type-passing vs. type-erasure semantics

In LI, there is no distinction between the use of constructors at compile time and run time. The consequence is that LI has a *type-passing* interpretation. Execution of LI programs depends on the typing annotations through the type constructor arguments to *typerec*.

This semantics is different from that of most conventional statically-typed languages. In a *type-erasure* semantics, run-time execution is modeled entirely by the term language. These languages have the property that execution will be the same even if all type information (such as the bindings for function arguments, type abstraction and type applications) is removed. This separation between compile-time computation (the equational theory at the type level) and run-time execution is known as a phase distinction. Typechecking a term does not depend on its run-time behavior[1], and the run-time behavior of a term does not depend on its compile-time type. Consequently, it is not obvious how to implement run-time type analysis in a language with a type-erasure semantics as types are not allowed to affect run-time execution.

However, although a type-passing semantics provides a concise and elegant way to specify run-time type analysis it is undesirable for two reasons.

First, the operational semantics of LI always constructs and passes type information to polymorphic functions, even when it is not necessary or desirable. Passing type information at run time incurs a run-time cost. A type-passing framework

[1]Dependently typed languages [Aug98, Bar92, BBC$^+$97, CAB$^+$86] are an example where typechecking a term may depend on its run-time value.

cannot express the optimization of eliminating unexamined types in order to improve performance. In addition, for reasons of modularity, it may be desirable to withhold run-time type information from a function to enforce type abstraction. In conventional type systems, abstraction may be implemented by hiding the identity of types either through parametric polymorphism [Rey83] or through existential types [MP88]. However, when types may be analyzed, the identity of types cannot be hidden so abstraction is impossible. For example, consider the type $\exists\alpha.\alpha$. In LI, this type implements a *dynamic* type; an expression of this type provides an object of some unknown type, and that unknown type's identity can be determined at run time by analyzing α, as in the cast example of Section 2.1.3. In a type-erasure system, $\exists\alpha.\alpha$ implements a useless abstract type because the identity of α cannot be determined.

Furthermore, the goal of typed low-level languages is to describe precisely the operation of real machines [MWCG99]. For example, a low-level language may describe the allocation behavior of the program or make register usage explicit. However, with the lack of phase distinction, both terms and type constructors describe run-time execution. Therefore, the semantics of the language must duplicate language constructs that describe low-level execution. For example, in a semantics that makes memory allocation explicit [MFH95, MH97], all data must be stored in an explicit heap. A type passing semantics must include forms for storing and retrieving types as well as terms from memory. Another particularly important example that occurs during type-directed compilation is closure conversion. As described by Morrisett *et al.* [MWCG99], in a type-erasure language, the partial application of a polymorphic function to a type may be considered a value as the application has no run-time significance. Therefore, closed code may simply be instantiated with its type environment when a closure is created. However, in a type-passing framework, the instantiation with a type environment can have some run-time effect. Therefore, in the result of closure conversion in the context of a type-passing language [MMH96], this type application must be delayed until the function is invoked. Describing this delay requires the addition of complicated mechanisms (including abstract kinds and translucent types) to create a closure's type environment.

A possible solution to the first problem (but not the second) would be to introduce a phase distinction between type constructors: Those purely necessary for type checking would be marked static and the remainder dynamic, with restrictions prohibiting dynamic type information from depending on static type constructors.[2] A possible solution to the second problem (but not the first) would be to combine

[2] A framework of how to construct such a language appears in the DCC work of Abadi et al. [ABHR99] or in the two-level type systems supporting partial evaluation [NN92].

the type and term languages together in the same syntactic class, as in Pure Type Systems [Bar92]. However, then the constructs used to describe run-time execution would also complicate (and likely prevent the decidability of [Aug98]) compile-time type checking.

In typed compilation, we need a language with a *type-erasure* semantics. In this chapter, I describe how type analysis may be modeled by the language λ_R of Crary, Weirich and Morrisett [CWM02]. This language implements type analysis by introducing special terms that represent the run-time types. I also describe the process of *phase splitting* or separating the compile-time objects from those at run time, by providing an embedding of LI into the type-erasure language LIR.

3.2 Term representations of types

For comparison, I present the LIR language as an extension of LI and focus on the differences. The principal difference between the two languages is the introduction of terms that represent types and the restriction of type analysis to those types for which representations are provided. This change does not diminish the expressiveness of LIR; LI may be translated in a straightforward manner into LIR, described in Section 3.4.

As an extension of LI, the LIR language is defined by the same judgments for the static and operational semantics. To emphasize that a typing judgment is specifically for the LIR language, we use $\vdash_R$ when it is not clear from context. Also, the notation $\mapsto_R$ refers to the operational semantics of LIR.

The new and modified syntactic forms of LIR are shaded in Table 3.1. The kind and constructor language of LIR is identical to that of LI. The key additions to the term language of LIR are representations of the basic type constructors. For example, the constructors *int* and $\rightarrow$ are represented by the new terms R_{int} and $R_{\rightarrow}$. We can create representations of any type-constructor using these terms: for example, $int \rightarrow int$ is represented by the term

$$(((R_{\rightarrow}[int])\, R_{int})[int])\, R_{int} = R_{\rightarrow}[int]\, R_{int}[int]\, R_{int}\,.$$

So that we may know what type constructor a term represents, the constructor is part of the term's type. The LIR language includes a new type $R(\tau)$ to describe the representation of the constructor τ. The types of the representations depend on the kinds of the constructors they represent, in a schema similar to that of the last section in Table 2.6. The representation of a constructor c of kind κ has a type inductively defined by κ, shown in Table 3.2. For example, the constant R_{int} has type $R\langle int : \star\rangle = R(int)$, and the constant $R_{\rightarrow}$ has type

$$R\langle \rightarrow : \star \rightarrow \star \rightarrow \star\rangle = \forall\alpha{:}\,\star\,.R(\alpha) \rightarrow \forall\beta{:}\,\star\,.R(\beta) \rightarrow R(\alpha \rightarrow \beta)$$

Table 3.1: LIR: Syntax

(kinds)	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2$
(con's)	c	$::=$	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid int \mid \to \mid \times$
		$\mid$	$Typerec\ c\,(c_{int}, c_{\to}, c_{\times})$
(types)	σ	$::=$	$T(c) \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall\alpha{:}\kappa.\sigma \mid \exists\alpha{:}\kappa.\sigma$
		$\mid$	$R(c)$
(terms)	e	$::=$	$i \mid x \mid \lambda x{:}\sigma.e \mid fix\ f{:}\sigma.e \mid e_1 e_2$
		$\mid$	$\langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e$
		$\mid$	$\Lambda\alpha{:}\kappa.v \mid e[c]$
		$\mid$	$pack\ \langle c, e\rangle\ as\ \exists\alpha{:}\kappa.\sigma \mid unpack\langle\alpha, x\rangle = e_1\ in\ e_2$
		$\mid$	$R_{int} \mid R_{\to} \mid R_{\times} \mid typerec[\alpha.\sigma]\ e\ \bar{e}$
(representation values)	v_r	$::=$	$R_{int} \mid R_{\to} \mid R_{\times} \mid v_r[c] \mid v_r\ v$
(values)	v	$::=$	$i \mid \lambda x{:}\sigma.e \mid fix\ f{:}\sigma.v \mid \langle v_1, v_2\rangle$
		$\mid$	$\Lambda\alpha{:}\kappa.v \mid (fix\ f{:}\sigma.v)[c_1]\ldots[c_n]$
		$\mid$	$pack\langle c, v\rangle\ as\ \exists\alpha{:}\kappa.\sigma$
		$\mid$	v_r

Table 3.2: LIR: Representation types

$$R\langle \tau : \star\rangle = R(\tau)$$
$$R\langle c : \kappa_1 \to \kappa_2\rangle = \forall\alpha{:}\kappa_1.R\langle \alpha : k_1\rangle \to R\langle c\alpha : \kappa_2\rangle$$

This representation of the arrow constructor guarantees that if e_1 and e_2 represent constructors c_1 and c_2, then the term $R_{\to}[c_1]e_1[c_2]e_2$ will represent the type constructor $c_1 \to c_2$.

In LIR, representations are analyzed instead of actual type constructors. That way, we are free to erase the type constructors before execution. The intuition is that whenever a type constructor is used, a corresponding type representation is

Table 3.3: LIR: Operational semantics of *typerec*

[*ev-trec-int*]	$\dfrac{}{\mathit{typerec}[\alpha.\sigma]\ \ R_{int}\ \ \overline{e} \mapsto_R e_{int}}$
[*ev-trec-arrow*]	$\dfrac{}{\begin{array}{l}\mathit{typerec}[\alpha.\sigma]\ (R_{\rightarrow}[c_1]\ v_1\ [c_2]\ v_2)\ \overline{e} \mapsto_R \\ \quad e_{\rightarrow}[c_1]\ v_1\ (\mathit{typerec}[\alpha.\sigma]\ v_1\ \overline{e})\ [c_2]\ v_2\ (\mathit{typerec}[\alpha.\sigma]\ v_2\ \overline{e})\end{array}}$
[*ev-trec-prod*]	$\dfrac{}{\begin{array}{l}\mathit{typerec}[\alpha.\sigma](R_{\times}[c_1]v_1[c_2]v_2)\ \overline{e}\ \ \mapsto_R \\ \quad e_{\times}[c_1]\ v_1\ (\mathit{typerec}[\alpha.\sigma]\ v_1\ \overline{e})\ [c_2]\ v_2\ (\mathit{typerec}[\alpha.\sigma]\ v_2\ \overline{e})\end{array}}$
[*ev-trec-cong*]	$\dfrac{e \mapsto_R e'}{\mathit{typerec}[\alpha.\sigma]\ e\ \overline{e} \mapsto_R \mathit{typerec}[\alpha.\sigma]\ e'\ \overline{e}}$

also supplied. Accordingly, the argument to the term level *typerec* is a term (representing some type constructor). For example, a *typerec* on the term $R_{\rightarrow}[c_1]\ v_1\ [c_2]\ v_2$ will step to the $R_{\rightarrow}$ branch, providing that branch with not only the type arguments c_1 and c_2 (as in LI), but also the representations of those arguments v_1 and v_2. This operation is part of the rules of the dynamic semantics, which appear in Table 3.3. Note that unlike LI, the rules *(ev-trec-int)*, *(ev-trec-arrow)*, and *(ev-trec-prod)* do not depend on the identity of types.

Because the type constructor represented by the argument is known during type checking, we can use it in the formation rules for *typerec* and *typecase*. However, the branches to *typerec* require more arguments. Therefore, we need a new schema to describe their types, notated by $|[\alpha.\sigma]\langle c : \kappa\rangle|$.

$$
\begin{array}{l}
|[\alpha.\sigma]\langle c : \star\rangle| = \sigma[c/\alpha] \\
|[\alpha.\sigma]\langle c : \kappa_1 \rightarrow \kappa_2\rangle| = \forall\alpha{:}\kappa_1.R\langle\alpha : \kappa_1\rangle \rightarrow |[\alpha.\sigma]\langle\alpha : \kappa_1\rangle| \rightarrow |[\alpha.\sigma]\langle c\alpha : \kappa_2\rangle|
\end{array}
$$

With this schema, the typing judgment for LIR's *typerec* is only a small modification to that of LI: We replace *typerec*'s argument c by its representation (which is of type $R(c)$). This judgment appears in Table 3.4.

The most important property of the dynamic semantics of LIR is that it permits type erasure. As types cannot influence run-time computation, implementers of are free to replace LIR with an analogous language, without typing annotations. To provide this erasure property, LIR must impose a value restriction on type abstractions. Without this restriction, a type abstraction (necessarily a value) could erase to a non-value, and then the erased language would not correctly simulate LIR. Furthermore, we must change the operational semantics of *fix*. If $(\mathit{fix}\ f{:}\sigma.v)[c]$ stepped to $v[fix f{:}\sigma.v/f][c]$ (as in LI), then a term in LIR would take a step, where its erasure $(\mathit{fix}\ f{:}\sigma.v)$ would not. For this reason $(\mathit{fix}\ f{:}\sigma.v)[c]$ is a value in LIR. An alternative would be to define the erasure of LIR such that type abstractions "erase" to value abstractions, and type applications erase to applications to the term unit. However, since it introduces computation, this unorthodox type erasure does not faithfully model the run-time execution.

Table 3.4: LIR: Static semantics

$\boxed{\Delta \vdash \sigma}$ **Type Formation**

[*ty-rep*]
$$\frac{\Delta \vdash_R c : \star}{\Delta \vdash_R R(c)}$$

$\boxed{\Delta \vdash \sigma = \sigma'}$ **Type Equivalence**

[*tyeq-rep*]
$$\frac{\Delta \vdash_R c = c' : \star}{\Delta \vdash_R R(c) = R(c')}$$

$\boxed{\Delta; \Gamma \vdash e : \sigma}$ **Term Formation**

[*e-rint*]
$$\frac{}{\Delta; \Gamma \vdash_R R_{\mathit{int}} : R\langle \mathit{int} : \star \rangle}$$

[*e-rarr*]
$$\frac{}{\Delta; \Gamma \vdash_R R_{\rightarrow} : R\langle \rightarrow : \star \rightarrow \star \rightarrow \star \rangle}$$

[*e-rprod*]
$$\frac{}{\Delta; \Gamma \vdash_R R_{\times} : R\langle \mathit{int} : \star \rightarrow \star \rightarrow \star \rangle}$$

[*e-tcase*]
$$\frac{\begin{array}{c}\Delta \vdash_R c : \star \\ \Delta; \Gamma \vdash_R e : R(c) \\ \Delta, \alpha{:}\star \vdash_R \sigma \\ \Delta; \Gamma \vdash_R e_{\mathit{int}} : |[\alpha.\sigma]\langle \mathit{int} : \star \rangle| \\ \Delta; \Gamma \vdash_R e_{\rightarrow} : |[\alpha.\sigma]\langle \rightarrow : \star \rightarrow \star \rightarrow \star \rangle| \\ \Delta; \Gamma \vdash_R e_{\times} : |[\alpha.\sigma]\langle \times : \star \rightarrow \star \rightarrow \star \rangle|\end{array}}{\Delta; \Gamma \vdash_R \mathit{typerec}[\alpha.\sigma]\, e\, (e_{\mathit{int}}, e_{\rightarrow}, e_{\times}) : \sigma[c/\alpha]}$$

3.2.1 A quick example

As an example of the use of LIR, we translate the *tostring* function from the previous section by requiring it to take an additional term argument, x_α, the representation of the argument type, α. The *typecase* term analyzes this representation instead of α, but otherwise, the function is the same.

$$
\begin{array}{l}
\mathit{fix}\ \mathit{tostring} : (\forall\alpha{:}\star.\ R(\alpha) \to \alpha \to \mathit{string}).\\
\quad \Lambda\alpha{:}\star.\ \lambda x_\alpha{:}R(\alpha).\\
\qquad \mathit{typecase}[\lambda\alpha{:}\star.\alpha \to \mathit{string}]\ x_\alpha\ \mathit{of}\\
\qquad\quad R_{\mathit{int}} \Rightarrow \mathit{int2string}\\
\qquad\quad R_{\to} \Rightarrow \Lambda\beta.\ \lambda x_\beta{:}R(\beta).\Lambda\gamma.\ \lambda x_\gamma{:}R(\gamma).\\
\qquad\qquad \lambda \mathit{obj}{:}\beta \to \gamma.\ \text{"function"}\\
\qquad\quad R_{\times} \Rightarrow \Lambda\beta.\ \lambda x_\beta{:}R(\beta).\Lambda\gamma.\ \lambda x_\gamma{:}R(\gamma).\\
\qquad\qquad \lambda \mathit{obj}{:}\beta \times \gamma.\\
\qquad\qquad\quad \texttt{"<"}\,\hat{}\,(\mathit{tostring}\,[\beta]\ x_\beta\ (\pi_1\,\mathit{obj}))\,\hat{}\\
\qquad\qquad\quad \texttt{","}\,\hat{}\,(\mathit{tostring}\,[\gamma]\ x_\gamma\ (\pi_2\,\mathit{obj}))\,\hat{}\,\texttt{">"}
\end{array}
$$

Figure 3.1: Example: *tostring* in LIR

3.3 Typing properties of LIR

Like the language LI, the LIR language possesses a number of important properties including decidability of type checking and type safety. In this section, I briefly cover the proofs of these properties. These proofs are not at all difficult to establish: each is an extension of the proof of the same property for LI.

Theorem 3.3.1 (Decidability of LIR type checking) *Given well-formed $\Delta;\Gamma$ and expression e, there is an algorithm to determine whether there exists a σ such that $\Delta;\Gamma \vdash e : \sigma$ is derivable in LIR.*

Because LIR shares the kinds and constructors of LI, results for that language still apply. The proof of decidability of LIR type checking is an extension of the decidability of LI type checking to a few new constructs. Again, the proof consists of two parts: showing that we may reduce types to a normal form, and showing that we may normalize type derivations to an equivalent syntax-directed version. Reduction of LIR types to normal form is very similar to that of LI types. The only difference is that we must normalize constructors appearing in R types. We may also produce normal forms for derivations in the same manner as LI.

Next, we would like to show that the static semantics guarantees safety. As in the last chapter (in Section 2.5), we prove type safety syntactically, in the manner popularized by Wright and Felleisen [WF94], by showing the usual Progress and Subject Reduction Lemmas.

We first show that, as before, we may substitute for type and term variables in all of the judgment forms.

Lemma 3.3.2 (Substitution) *1. If $\Delta, \alpha{:}\kappa' \vdash c : \kappa$ and $\Delta \vdash c' : \kappa'$ then $\Delta \vdash c[c'/\alpha] : \kappa$.*

2. If $\Delta, \alpha{:}\kappa' \vdash c_1 = c_2 : \kappa$ and $\Delta \vdash c' : \kappa'$ then $\Delta \vdash c_1[c'/\alpha] = c_2[c'/\alpha] : \kappa$.

3. If $\Delta, \alpha{:}\kappa \vdash \sigma$ and $\Delta \vdash c : \kappa$ then $\Delta \vdash \sigma[c/\alpha]$.

4. If $\Delta, \alpha{:}\kappa \vdash \sigma = \sigma'$ and $\Delta \vdash c : \kappa$ then $\Delta \vdash \sigma[c/\alpha] = \sigma'[c/\alpha]$.

5. If $\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$ and $\emptyset \vdash c : \kappa$ then $\Delta; \Gamma[c/\alpha] \vdash e[c/\alpha] : \sigma[c/\alpha]$.

6. If $\Delta; \Gamma, x{:}\sigma' \vdash e : \sigma$ and $\emptyset \vdash e' : \sigma'$ then $\Delta; \Gamma \vdash e[e'/x] : \sigma$.

The lemmas for the constructor language (parts 1 and 2) follow from the same results for LI, as the constructor language is unchanged. To prove substitution for the type and term language (parts 3-6), we extend the LI proofs with cases for the new constructs. These cases follow in a straightforward manner.

Lemma 3.3.3 (Subject Reduction) *If $\emptyset \vdash e : \tau$ and $e \mapsto e'$ then $\emptyset \vdash e' : \tau$.*

Proof of the Subject Reduction Lemma (3.3.3)

The proof of the Subject Reduction Lemma is by induction on the normalized derivation of $\emptyset \vdash e : \sigma$, with a case analysis on the last step of the derivation. Most of the cases are the same as the proof of Subject Reduction for LI (as in the previous chapter). The exception is the new case for *typerec* below:

case (term-trec) If the last rule applied in the derivation was the typerec rule, then the operational step taken depends on the argument to the typecase e. If the argument is not a value, then it steps to some e', and by induction we may conclude e' has the same type as e. We may use this result to conclude that a *typerec* on e' will also have the same type as before.

Otherwise, the argument is a value of type $R(c)$, and it must be one of R_{int}, $R_{\rightarrow}[c_1]v_1[c_2]v_2$, or $R_{\times}[c_1]v_1[c_2]v_2$ to take a step. If it is R_{int}, then c is equivalent to *int* and the typecase term is of type $\sigma[int/\alpha]$. By the operational semantics rule *(ev-trec-int)*, the *typerec* term steps to e_{int}, by assumption of type $\sigma[int/\alpha]$.

Otherwise, if it is $R_{\rightarrow}$, then the term steps to $e_{\rightarrow}[c_1]v_1[c_2]v_2$, by the rule *(ev-trec-arrow)* of the operational semantics. By assumption, with context $\emptyset$, the term $e_{\rightarrow}$ is of type $|[c]\langle\!\langle \rightarrow : \star \rightarrow \star \rightarrow \star \rangle\!\rangle|$ which expands to

$$\forall\beta{:}\star.R(\beta) \rightarrow \sigma[\beta/\alpha] \rightarrow \forall\gamma{:}\star.R(\gamma) \rightarrow \sigma[\gamma/\alpha] \rightarrow \sigma[(\beta \rightarrow \gamma)/\alpha].$$

Therefore, we may construct the desired judgment of the well formedness of the term with repeated use of *(term-app)* and *(term-tapp)*, and additional derivations of well formedness of the type constructor and term arguments.

$$\begin{array}{l}
\emptyset \vdash c_1 : \star \\
\emptyset \vdash v_1 : R(c_1) \\
\emptyset \vdash c_2 : \star \\
\emptyset \vdash v_2 : R(c_2) \\
\emptyset \vdash typerec[\alpha.\sigma]\ v_1\ (e_{int}, e_{\rightarrow}, e_{\times}) : \sigma[c_1/\alpha] \\
\emptyset \vdash typerec[\alpha.\sigma]\ v_2\ (e_{int}, e_{\rightarrow}, e_{\times}) : \sigma[c_2/\alpha]
\end{array}$$

The first four of these judgments may be derived from inversion, as we must have derived $\emptyset \vdash R_{\rightarrow}[c_1]v_1[c_2]v_2 : R(c_1 \rightarrow c_2)$ to apply *(term-trec)*. The last two judgments may be derived from the first four and judgments about σ, e_{int}, $e_{\rightarrow}$, and $e_{\times}$ necessary for *(term-trec)*.

Finally, if the argument to *typerec* is $R_{\times}[c_1]v_1[c_2]v_2$, then the case is symmetric to the case above for $R_{\rightarrow}$.

□

To extend LI's Progress Lemma (2.5.7) to LIR, we need to extend the Canonical Forms Lemma (2.5.5) to include the representation types. Again, this lemma tells us that the form of a closed value is determined by its type.

Lemma 3.3.4 (Canonical Forms) *If* $\emptyset \vdash v : \sigma$ *then*

1. *If* $\emptyset \vdash \sigma = int$ *then* v *is* i.

2. *If* $\emptyset \vdash \sigma = \sigma_1 \rightarrow \sigma_2$ *then* v *is either* $\lambda x{:}\sigma_1.e$ *or* $(fix\ f{:}(\sigma_1 \rightarrow \sigma_2).v')[c_1]\cdots[c_n]$.

3. *If* $\emptyset \vdash \sigma = \sigma_1 \times \sigma_2$ *then* v *is of the form* $\langle v_1, v_2\rangle$.

4. *If* $\emptyset \vdash \sigma = \forall\alpha{:}\kappa.\sigma$ *then* v *is either* $\Lambda\alpha{:}\kappa.v'$ *or* $(fix\ f{:}(\forall\alpha{:}\kappa.\sigma).v')[c_1]\cdots[c_n]$.

5. *If* $\emptyset \vdash \sigma = \exists\alpha{:}\kappa.\sigma$ *then* v *is pack* v' *as* $\exists\alpha.\sigma$ *hiding* σ'.

6. *If* $\emptyset \vdash \sigma = R(int)$ *then* v *is* R_{int}.

7. *If* $\emptyset \vdash \sigma = R(c_1 \to c_2)$ *then* v *is of the form* $R_{\to}[c_1']v_1[c_2']v_2$, *where* $\emptyset \vdash c_1 = c_1' : \star$ *and* $\emptyset \vdash c_2 = c_2' : \star$.

8. *If* $\emptyset \vdash \sigma = R(c_1 \times c_2)$ *then* v *is of the form* $R_{\times}[c_1']v_1[c_2']v_2$, *where* $\emptyset \vdash c_1 = c_1' : \star$ *and* $\emptyset \vdash c_2 = c_2' : \star$.

Proof

Proof follows from examination of the normal derivations that produce values. □

Lemma 3.3.5 (Progress) *If* $\emptyset \vdash e : \tau$ *and* e *is not a value then there exists an* e' *such that* $e \mapsto e'$.

Proof of the Progress Lemma (3.3.5)

Proof of the Progress Lemma is by induction on the derivation of $\emptyset \vdash e : \tau$, with a case analysis on the last rule applied in the derivation. We present the case for *typerec* below:

case (term-trec) If the argument e to *typerec* is not a value, then by induction, it steps to e', so the entire term steps to a *typerec* on e' with the same branches. Otherwise, if it is a value, by inversion of the formation rule for *typerec*, e must be of type $R(c)$ for some constructor c. Furthermore, by Lemma 2.5.6, as c is closed and of kind $\star$, c must be equivalent to either int, $c_1 \to c_2$ or $c_1 \times c_2$ for some c_1 and c_2. Therefore, by canonical forms, e is either R_{int}, $R_{\to}[c_1']v_1[c_2']v_2$ or $R_{\times}[c_1']v_1[c_2']v_2$, as these are the only values of the appropriate type. For each of these values there is a corresponding rule in the operational semantics (Figure 3.3).

□

Because we have proven preservation and progress for LIR, then we may prove LIR type safety in the same manner as LI type safety.

Theorem 3.3.6 (LIR Type Soundness) *If* $\emptyset \vdash e : \sigma$ *and* $e \mapsto^* e'$ *then* e' *is not stuck.*

Proof

See Theorem 2.5.8. □

Table 3.5: Translation of LI types and terms

types

$$
\begin{aligned}
|T(c)| &= c \\
|\sigma_1 \to \sigma_2| &= |\sigma_1| \to |\sigma_2| \\
|\sigma_1 \times \sigma_2| &= |\sigma_1| \times |\sigma_2| \\
|\forall\alpha{:}\kappa.\sigma| &= \forall\alpha{:}\kappa.R\langle\alpha : \kappa\rangle \to |\sigma| \\
|\exists\alpha{:}\kappa.\sigma| &= \exists\alpha{:}\kappa.R\langle\alpha : \kappa\rangle \times |\sigma|
\end{aligned}
$$

expressions

$$
\begin{aligned}
|x| &= x \\
|i| &= i \\
|\lambda x{:}\sigma.e| &= \lambda x{:}|\sigma|.|e| \\
|\mathit{fix}\ f{:}\sigma.e| &= \mathit{fix}\ f{:}|\sigma|.|e| \\
|e_1 e_2| &= |e_1||e_2| \\
|\langle e_1, e_2\rangle| &= \langle |e_1|, |e_2|\rangle \\
|\pi_1 e| &= \pi_1|e| \\
|\pi_2 e| &= \pi_2|e| \\
|\Lambda\alpha{:}\kappa.e| &= \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.|e| \\
|e[c]| &= |e|\,[c]\,\mathcal{R}|c| \\
|\,\mathit{pack}\ \langle c, e\rangle\ \mathit{as}(\exists\alpha{:}\kappa.\sigma)| &= \mathit{pack}\ \langle c, \langle\mathcal{R}|c|, |e|\rangle\rangle \\
&\qquad \mathit{as}\ \exists\alpha{:}\kappa.R\langle\alpha : \kappa\rangle \times |\sigma| \\
|\,\mathit{unpack}\langle\alpha, x\rangle = e_1\ \mathit{in}\ e_2| &= \mathit{unpack}\langle\alpha, y\rangle = |e_1| \\
&\qquad \mathit{in}\ (\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle. \\
&\qquad\quad \lambda x : \alpha.|e_2|)(\pi_1 y)(\pi_2 y) \\
|\,\mathit{typerec}[\alpha.\sigma]\ c\ (e_{int}, e_\to, e_\times)| &= \mathit{typerec}[\alpha.|\sigma|]\,\mathcal{R}|c| \\
&\qquad (|\,e_{int}\,|, |\,e_\to\,|, |\,e_\times\,|)
\end{aligned}
$$

3.4 Embedding of LI

I next formalize the connection between LI and LIR by showing how any code written in LI may be expressed in LIR. In this section, I describe a translation (written $|\cdot|$) of LI expressions into LIR. The full details of this embedding appear in Tables 3.5 and 3.6. I include this embedding for two reasons: first, to show that LIR is as expressive as LI, and second, to demonstrate a simple use of LIR as an intermediate language. The main difference between LI and LIR is the *typerec* term; in LI, it takes a type constructor as its argument, in LIR, it takes a term representing a type. Therefore, to simulate a LI *typerec* term with an LIR *typerec*

Table 3.6: Translation of LI constructors

$$
\begin{array}{rcl}
\mathcal{R}|\,\mathit{int}\,| & = & R_{\mathit{int}} \\
\mathcal{R}|\,\to\,| & = & R_{\to} \\
\mathcal{R}|\,\times\,| & = & R_{\times} \\
\mathcal{R}|\alpha| & = & x_\alpha \\
\mathcal{R}|\lambda\alpha{:}\kappa.c| & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha:\kappa\rangle.\mathcal{R}|c| \\
\mathcal{R}|c_1c_2| & = & \mathcal{R}|c_1|[c_2]\mathcal{R}|c_2| \\
\mathcal{R}|\,\mathit{Typerec}\ c\ (c_{\mathit{int}}, c_{\to}, c_{\times})| & = & \mathit{typerec}[\alpha.R\langle \mathit{rec}(\alpha):\kappa\rangle]\ \mathcal{R}|c| \\
 & & \quad R_{\mathit{int}} \Rightarrow \mathcal{R}|\,c_{\mathit{int}}\,| \\
 & & \quad R_{\to} \Rightarrow \mathit{expand}(\mathcal{R}|\,c_{\to}\,|) \\
 & & \quad R_{\times} \Rightarrow \mathit{expand}(\mathcal{R}|\,c_{\times}\,|) \\
 & & \\
\text{where } \mathit{expand}(e) & = & \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha).\lambda y_\alpha{:}R\langle \mathit{rec}(\alpha):\kappa\rangle. \\
 & & \quad \Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).\lambda y_\beta{:}R\langle \mathit{rec}(\beta):\kappa\rangle. \\
 & & \quad\quad e\ [\alpha]\ x_\alpha\ [\mathit{rec}(\alpha)]\ y\ [\beta]\ x_b\ [\mathit{rec}(\beta)]\ y \\
 & & \\
\text{and } \mathit{rec}(\alpha) & = & \mathit{Typerec}\ \ \alpha\ (c_{\mathit{int}}, c_{\to}, c_{\times})
\end{array}
$$

term, we must be able to form the term representation of the type constructor argument. This operation, written $\mathcal{R}|\cdot|$, appears in Table 3.6.

Creating the representation of a given type constructor is complicated by the fact that the argument to *Typerec* may contain constructors with free type variables. These type variables are translated to term variables that represent them, but we need to maintain the invariant that for every accessible type variable, a corresponding term variable representing it is also accessible. We make this guarantee by a process reminiscent of "phase splitting" [HMM90] or evidence passing [Jon92]. In the translation of constructor abstractions (at both the constructor and term level), we split the abstractions to take both a constructor and a term variable, where the term variable must be the representation of that constructor. We also change application accordingly. This translation satisfies the value restriction placed on LIR type abstractions as term abstractions follow all type abstractions. Dually, we also include the representation of a type constructor when we form an existential package.

Table 3.7: Translation of LI contexts

$$
\begin{array}{rcl}
R_{\mathsf{val}}(\emptyset) & = & \emptyset \\
R_{\mathsf{val}}(\Delta, \alpha{:}\kappa) & = & R_{\mathsf{val}}(\Delta), x_\alpha{:}R\langle \alpha : \kappa \rangle \\
\\
|\emptyset| & = & \emptyset \\
|\Gamma, x{:}\sigma| & = & |\Gamma|, x{:}|\sigma| \\
\\
|\Delta; \Gamma| & = & \Delta; R_{\mathsf{val}}(\Delta), |\Gamma|
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{R}|\alpha| & = & x_\alpha \\
\mathcal{R}|\lambda\alpha{:}\kappa.c| & = & \Lambda\alpha{:}\kappa.\, \lambda x_\alpha{:}R\langle \alpha : \kappa \rangle.\, \mathcal{R}|c| \\
\mathcal{R}|c_1 c_2| & = & \mathcal{R}|c_1|\,[c_2]\,\mathcal{R}|c_2| \\
\\
|\Lambda\alpha{:}\kappa.e| & = & \Lambda\alpha{:}|\kappa|.\lambda x_\alpha{:}R\langle \alpha : \kappa \rangle.|e| \\
|e[c]| & = & |e|\,[c]\,\mathcal{R}|c|
\end{array}
$$

Given a type variable, α, what is the type of its corresponding term variable, x_α? If α is of kind $\star$, then x_α should be of type $R(\alpha)$. If α is of a higher kind, say, for example, a function from types to types, then x_α should map type representations to type representations and its type should reflect that fact. For this reason, to constrain the type of x_α we use $R\langle c : \kappa \rangle$, the type of the representations of constructor c with kind κ. If the constructor c is of kind $\kappa_1 \to \kappa_2$, its representation is a polymorphic function that takes the representation of the argument constructor to the representation of the result of applying c to that argument.

The last part of the translation of type constructors to their representations is the definition of the representation of a *Typerec* constructor. We represent it as a *typerec* on the representation of the argument to the *Typerec*.

3.4.1 Properties of the embedding

In this Section, I show the static and dynamic correctness of the embedding.

The first lemma states that if a type constructor is well formed, then so is the type of its representation.

Lemma 3.4.1 *If* $\Delta \vdash c : \kappa$, *then* $\Delta \vdash R\langle c : \kappa \rangle$

Proof

by induction on k. If $\kappa{=}\star$, $\Delta \vdash R(c)$. If $\kappa = \kappa_1 \to \kappa_2$, then, as $\Delta, \alpha{:}\kappa \vdash \alpha{:}\kappa_1$ then by induction $\Delta, \alpha{:}\kappa_1 \vdash R\langle \alpha : \kappa_1 \rangle$ and as $\Delta, \alpha{:}\kappa_1 \vdash c\alpha : \kappa_2$, then by induction, $\Delta, \alpha{:}\kappa_1 \vdash R\langle c\alpha : \kappa_2 \rangle$. Therefore, $\Delta \vdash R\langle c : \kappa_1 \to \kappa_2 \rangle$. □

The following lemma states that the term representations have the correct type. For this lemma, we must construct an appropriate context to check the representation: one that contains term variables to represent every type variable in the context Δ. This operation $R_{\mathsf{val}}(\cdot)$ appears in Table 3.7.

Lemma 3.4.2 *If* $\Delta \vdash c : \kappa$ *then* $\Delta; R_{\mathsf{val}}(\Delta) \vdash_R \mathcal{R}|c| : R\langle c : \kappa \rangle$

Proof

Proof is by induction on $\Delta \vdash c : \kappa$. Selected cases are shown below.

case (con-var) Suppose $\Delta \vdash \alpha : \kappa$. Thus $\Delta = \Delta', \alpha{:}\kappa$, for some Δ'. Therefore

$$\Delta', \alpha{:}\kappa; R_{\mathsf{val}}(\Delta'), x_\alpha : R\langle \alpha : \kappa \rangle \vdash x_\alpha : R\langle \alpha : \kappa \rangle$$

case (con-fn) Suppose $\Delta \vdash \lambda\alpha{:}\kappa_1.c' : \kappa_1 \to \kappa_2$. By induction

$$\Delta, \alpha{:}\kappa_1; R_{\mathsf{val}}(\Delta, \alpha{:}\kappa_1) \vdash \mathcal{R}|c'| : R\langle c' : \kappa_1 \rangle.$$

Therefore,

$$\Delta; R_{\mathsf{val}}(\Delta) \vdash \Lambda\alpha{:}\kappa_1.\lambda x_\alpha{:}R\langle \alpha : \kappa_1 \rangle.\mathcal{R}|c'| : \forall\alpha{:}\kappa_1.R\langle \alpha : \kappa_1 \rangle \to R\langle c' : \kappa_1 \rangle,$$

from which we may conclude

$$\Delta; R_{\mathsf{val}}(\Delta) \vdash \mathcal{R}|\lambda\alpha{:}\kappa_1.c'| : R\langle \lambda\alpha{:}\kappa_1.c' : \kappa_1 \to \kappa_2 \rangle$$

case (con-trec) Suppose $\Delta \vdash \mathit{Typerec}\ c'\ (c_{int}, c_{\to}, c_{\times})$. Let the notation $(\mathit{rec}(c))$ be an abbreviation for the type constructor $\mathit{Typerec}\ c\ (c_{int}, c_{\to}, c_{\times})$. We need to show that

$$\begin{array}{lr}
\Delta; R_{\mathsf{val}}(\Delta) \vdash \mathit{typerec}[\alpha.R\langle \mathit{rec}(\alpha) : \kappa \rangle]\ \mathcal{R}|c| & : R\langle \mathit{rec}(c') : \kappa \rangle \\
\quad R_{int} \Rightarrow \mathcal{R}|\ c_{int}\ | & \\
\quad R_{\to} \Rightarrow \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha).\lambda y_\alpha{:}R\langle \mathit{rec}(\alpha) : \kappa \rangle. & \\
\qquad \Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).\lambda y_\beta{:}R\langle \mathit{rec}(\beta) : \kappa \rangle. & \\
\qquad\quad \mathcal{R}|\ c_{\to}\ |\ [\alpha]\ x_\alpha\ [\mathit{rec}(\alpha)]\ y\ [\beta]\ x_b\ [\mathit{rec}(\beta)]\ y & \\
\quad R_{\times} \Rightarrow \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha).\lambda y_\alpha{:}R\langle \mathit{rec}(\alpha) : \kappa \rangle. & \\
\qquad \Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).\lambda y_\beta{:}R\langle \mathit{rec}(\beta) : \kappa \rangle. & \\
\qquad\quad \mathcal{R}|\ c_{\times}\ |\ [\alpha]\ x_\alpha\ [\mathit{rec}(\alpha)]\ y\ [\beta]\ x_b\ [\mathit{rec}(\beta)]\ y &
\end{array}$$

To derive this judgment, we must satisfy the following preconditions:

1. $\Delta; R_{\mathsf{val}}(\Delta) \vdash \mathcal{R}|c'| : R(c')$
2. $\Delta; R_{\mathsf{val}}(\Delta) \vdash \mathcal{R}|\ c_{int}\ | : |[\alpha.R\langle rec(\alpha) : \kappa\rangle]\langle int : \star\rangle|$
3. $\Delta; R_{\mathsf{val}}(\Delta) \vdash \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha).\lambda y_\alpha{:}R\langle rec(\alpha) : \kappa\rangle.$
   $\Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).\lambda y_\beta{:}R\langle rec(\beta) : \kappa\rangle.$
   $\mathcal{R}|\ c_\to\ |\ [\alpha]\ x_\alpha\ [rec(\alpha)]\ y_\alpha\ [\beta]\ x_b\ [rec(\beta)]\ y_\beta$
   $: |[\alpha.R\langle rec(\alpha) : \kappa\rangle]\langle \to : \star \to \star \to \star\rangle|$
4. $\Delta; R_{\mathsf{val}}(\Delta) \vdash \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha).\lambda y_\alpha{:}R\langle rec(\alpha) : \kappa\rangle.$
   $\Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).\lambda y_\beta{:}R\langle rec(\beta) : \kappa\rangle.$
   $\mathcal{R}|\ c_\times\ |\ [\alpha]\ x_\alpha\ [rec(\alpha)]\ y_\alpha\ [\beta]\ x_b\ [rec(\beta)]\ y_\beta$
   $: |[\alpha.R\langle rec(\alpha) : \kappa\rangle]\langle \times : \star \to \star \to \star\rangle|$

The first follows immediately by induction. For the second, the type

$$|[\alpha.R\langle rec(\alpha) : \kappa\rangle]\langle int : \star\rangle| = R\langle rec(int) : \kappa\rangle = R\langle c_{int} : \kappa\rangle$$

so again, the result follows by induction. For the third, we can conclude by induction that

$$\Delta; R_{\mathsf{val}}(\Delta) \vdash \mathcal{R}|\ c_\to\ | : R\langle c_\to : \star \to \kappa \to \star \to \kappa \to \kappa\rangle$$

The type $R\langle c_\to : \star \to \kappa \to \star \to \kappa \to \kappa\rangle$ equals

$$\begin{aligned}&\forall\alpha_1{:}\star.R(\alpha_1) \to \forall\alpha_2{:}\kappa.R\langle\alpha_2 : \kappa\rangle \to \forall\beta_1{:}\star.R(\beta_1) \to \forall\beta_2{:}\kappa.R\langle\beta_2 : \kappa\rangle\\ &\quad \to R\langle c_\to\, \alpha_1\alpha_2\beta_1\beta_2 : \kappa\rangle\end{aligned}$$

so the application $\mathcal{R}|\ c_\to\ |\ [\alpha]\ x_\alpha\ [rec(\alpha)]\ y_\alpha\ [\beta]\ x_b\ [rec(\beta)]\ y_\beta$ is of type

$$R\langle c_\to\ \ \alpha\ [rec(\alpha)]\ \beta\ [rec(\beta)]\ \ : \kappa\rangle = R\langle rec(\alpha \to \beta) : \kappa\rangle$$

Therefore, by abstracting α, x_α, y_α, β, x_b, and y_β, we get a term of type

$$\begin{aligned}&\forall\alpha{:}\star.R(\alpha) \to R\langle rec(\alpha) : \kappa\rangle\\ &\quad \to \forall\beta{:}\star.R(\beta) \to R\langle rec(\beta) : \kappa\rangle \to R\langle rec(\alpha \to \beta) : \kappa\rangle\end{aligned}$$

which is the definition of

$$|[\alpha.R\langle rec(\alpha) : \kappa\rangle]\langle \to : \star \to \star \to \star\rangle|$$

Similar reasoning holds for the fourth precondition.

□

Theorem 3.4.3 (Static correctness) *1. If* $\Delta \vdash_i \sigma$ *then* $\Delta \vdash_R |\sigma|$

2. If $\Delta \vdash_i \sigma_1 = \sigma_2$ *then* $\Delta \vdash_R |\sigma_1| = |\sigma_2|$

3. If $\Delta; \Gamma \vdash_i e : \tau$ *then* $|\Delta; \Gamma| \vdash_R |e| : |\tau|$

Proof

Proof is by induction on derivations. Selected cases appear below:

case (con-all) Assume $\Delta \vdash_i \forall\alpha{:}\kappa.\sigma$. By induction, $\Delta, \alpha{:}\kappa \vdash_R |\sigma|$. Therefore $\Delta \vdash_R \forall\alpha{:}\kappa.R\langle\alpha : \kappa\rangle \to |\sigma|$.

case (term-trec) Assume $\Delta; \Gamma \vdash_i \mathit{typerec}[\alpha.\sigma]\ c\ (e_{int}, e_{\to}, e_{\times}) : \sigma[c/\alpha]$
By Lemma 3.4.1

$$\Delta; R_{\mathsf{val}}(\Delta) \vdash_R \mathcal{R}|c| : R(c).$$

By Part 1, $\Delta, \alpha{:}\star \vdash_R |\sigma|$. By induction

$$\begin{array}{l} |\Delta; \Gamma| \vdash_R |\ e_{int}\ | : |[\alpha.\sigma]\langle int : \star\rangle|, \\ |\Delta; \Gamma| \vdash_R |\ e_{\to}\ | : |[\alpha.\sigma]\langle \to : \star \to \star \to \star\rangle| \text{ and} \\ |\Delta; \Gamma| \vdash_R |\ e_{\times}\ | : |[\alpha.\sigma]\langle \times : \star \to \star \to \star\rangle|. \end{array}$$

Therefore,

$$|\Delta; \Gamma| \vdash_R \mathit{typerec}[\alpha.|\sigma|]\ \mathcal{R}|c|\ (|\ e_{int}\ |, |\ e_{\to}\ |, |\ e_{\times}\ |) : |\sigma[c/\alpha]|$$

□

Theorem 3.4.4 (Dynamic Correctness) *If* $e \mapsto_i^* v$ *then* $|e| \mapsto_R^* v'$ *and* $v' \equiv_{\mathcal{T}} |v|$.

In order to show the dynamic correctness of the embedding, we must show that the result of translation simulates the operation of LI. However, because the evaluation of the term representations does not exactly match the reduction of constructors, we must add some imprecision to the simulation. We allow constructors and their representations appearing in the result of the embedding to be of any equivalent constructor (based on the definition of constructor equality) instead of exactly matching the constructor appearing in the source LI term.

First, in Table 3.8, we define $[\![c]\!]$ as the set of all constructors equal to c. Using this set, we define the operation $\mathcal{R}[\![c]\!]$ that produces a set of representations of the constructor c. For any c, $\mathcal{R}|c|$ is in the set $\mathcal{R}[\![c]\!]$. The other members of this set differ from $\mathcal{R}|c|$ only the embedded constructors. For example, $\mathcal{R}[\![int \to int]\!]$ includes both $R_{\to}[int, int](R_{int}, R_{int})$, and $R_{\to}[(\lambda\beta{:}\star.\beta)\ int, int](R_{int}, R_{int})$. The set $\overline{\mathcal{R}[\![c]\!]}$,

Table 3.8: Extended representations

$$
\begin{array}{rcl}
[\![c]\!] & = & \{c' \mid \Delta \vdash c' = c : \kappa\} \\
\\
\mathcal{R}[\![int]\!] & = & \{R_{int}\} \\
\mathcal{R}[\![\rightarrow]\!] & = & \{R_{\rightarrow}\} \\
\mathcal{R}[\![\times]\!] & = & \{R_{\times}\} \\
\mathcal{R}[\![\alpha]\!] & = & \{x_\alpha\} \\
\mathcal{R}[\![\lambda\alpha{:}\kappa.c]\!] & = & \{\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.e \mid e \in \mathcal{R}[\![c]\!]\} \\
\mathcal{R}[\![c_1 c_2]\!] & = & \{e_1[c_2']e_2 \mid e_1 \in \mathcal{R}[\![c_1]\!], c_2' \in [\![c_2]\!], e_2 \in \mathcal{R}[\![c_2]\!]\} \\
\mathcal{R}[\![\mathit{Typerec}\ \tau(c_{int}, c_{\rightarrow}, c_{\times})]\!] & = & \left\{ \begin{array}{l|l} \mathit{typerec}[\alpha.R\langle rec(\alpha) : \kappa\rangle]\ e & e \in \mathcal{R}[\![\tau]\!] \\ \quad R_{int} \Rightarrow e_{int} & e_{int} \in \mathcal{R}[\![c_{int}]\!] \\ \quad R_{\rightarrow} \Rightarrow \mathit{expand}(e_{\rightarrow}) & e_{\rightarrow} \in \mathcal{R}[\![c_{\rightarrow}]\!] \\ \quad R_{\times} \Rightarrow \mathit{expand}(e_{\times}) & e_{\times} \in \mathcal{R}[\![c_{\times}]\!] \end{array} \right\} \\
\text{where } \mathit{expand}(e) & = & \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha).\lambda y_\alpha{:}R\langle rec(\alpha) : \kappa\rangle. \\
& & \quad \Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).\lambda y_\beta{:}R\langle rec(\beta) : \kappa\rangle. \\
& & \qquad e\ [\alpha]\ x_\alpha\ [rec(\alpha)]\ y\ [\beta]\ x_b\ [rec(\beta)]\ y \\
\text{and } rec(\alpha) & = & \mathit{Typerec}\ \ \alpha\ (c_{int}, c_{\rightarrow}, c_{\times}) \\
\\
\overline{\mathcal{R}[\![c]\!]} & = & \{e \mid c' \in [\![c]\!] \quad \& \quad e \in \mathcal{R}[\![c']\!]\}
\end{array}
$$

defined at the bottom of the table, is even larger. It includes all representations of equivalent constructors. For example, not only does $\overline{\mathcal{R}[\![int \rightarrow int]\!]}$ include the above terms, but it also includes a representation of $((\lambda\beta{:}\star.\beta)\ int) \rightarrow int$

$$R_{\rightarrow}[(\lambda\beta{:}\star.\beta)\ int, int]((\Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).x)\ R_{int}), R_{int}).$$

Likewise, the operations $[\![\sigma]\!]$ and $[\![e]\!]$ in Table 3.9 generalize the translation of LI types and terms. Again $|\sigma|$ is in the set $[\![\sigma]\!]$ and $|e|$ is in $[\![e]\!]$. In these sets, embedded constructors and their representations may be replaced with equivalent forms. For example, $[\![T(int)]\!]$ includes both the types $T(int)$ and $T((\lambda\beta{:}\star.\beta)\ int)$. For the translation of terms, $[\![x[int]]\!]$ includes $x[int]\ R_{int}$, $x[(\lambda\beta{:}\star.\beta)\ int]\ R_{int}$, and $x[int]((\Lambda\beta{:}\star.\lambda x_\beta{:}R(\beta).x)\ R_{int})$.

In terms of typing, all terms in $\mathcal{R}[\![c]\!]$ have the same typing properties as $\mathcal{R}|c|$:

Lemma 3.4.5 *If $\Delta \vdash c : \kappa$ then for all $e \in \mathcal{R}[\![c]\!]$, $\Delta; R_{\mathsf{val}}(\Delta) \vdash e : R\langle c : \kappa\rangle$.*

Proof sketch

Table 3.9: Extended translation

$$
\begin{array}{rcl}
\textit{types} & & \\
[\![T(c)]\!] & = & \{T(c') \mid c' \in [\![c]\!]\} \\
[\![int]\!] & = & \{int\} \\
[\![\sigma_1 \to \sigma_2]\!] & = & \{\sigma_1' \to \sigma_2' \mid \sigma_i \in [\![\sigma_i]\!],\ i = 1,2\} \\
[\![\sigma_1 \times \sigma_2]\!] & = & \{\sigma_1' \times \sigma_2' \mid \sigma_i \in [\![\sigma_i]\!],\ i = 1,2\} \\
[\![\forall\alpha{:}\kappa.\sigma]\!] & = & \{\forall\alpha{:}\kappa.R\alpha\kappa \to \sigma' \mid \sigma' \in [\![\sigma]\!]\} \\
[\![\exists\alpha{:}\kappa.\sigma]\!] & = & \{\exists\alpha{:}\kappa.R\alpha\kappa \times \sigma' \mid \sigma' \in [\![\sigma]\!]\} \\
 & & \\
\textit{expressions} & & \\
[\![x]\!] & = & \{x\} \\
[\![i]\!] & = & \{i\} \\
[\![\lambda x{:}\sigma.e]\!] & = & \{\lambda x{:}\sigma'.e' \mid \sigma' \in [\![\sigma]\!], e' \in [\![e]\!]\} \\
[\![\mathit{fix}\ f{:}\sigma.v]\!] & = & \{\mathit{fix}\ f{:}\sigma'.v' \mid \sigma' \in [\![\sigma]\!], v' \in [\![v]\!]\} \\
[\![e_1\ e_2]\!] & = & \{e_1'\ e_2' \mid e_1' \in [\![e_1]\!],\ e_2' \in [\![e_2]\!]\} \\
[\![\langle e_1, e_2\rangle]\!] & = & \{\langle e_1', e_2'\rangle \mid e_1' \in [\![e_1]\!],\ e_2' \in [\![e_2]\!]\} \\
[\![\pi_1 e]\!] & = & \{\pi_1 e' \mid e' \in [\![e]\!]\} \\
[\![\pi_2 e]\!] & = & \{\pi_2 e' \mid e' \in [\![e]\!]\} \\
[\![\Lambda\alpha{:}\kappa.e]\!] & = & \{\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha:\kappa\rangle.e' \mid e' \in [\![e]\!]\} \\
[\![e[c]]\!] & = & \{e'[c']\ e'' \mid e' \in [\![e]\!], c' \in [\![c]\!], e'' \in \overline{\mathcal{R}[\![c]\!]}\} \\
\left[\!\!\left[\begin{array}{l} \mathit{pack}\ e\ \mathit{as}(\exists\alpha{:}\kappa.\sigma) \\ \quad \mathit{hiding}\ c \end{array} \right]\!\!\right] & = & \left\{ \begin{array}{l|l} \mathit{pack}\ \langle e'', e'\rangle & e'' \in \overline{\mathcal{R}[\![c]\!]} \\ \quad \mathit{as}\ \exists\alpha{:}\kappa.\ R\langle\alpha:\kappa\rangle \times \sigma' & e' \in [\![e]\!], \sigma' \in [\![\sigma]\!] \\ \quad \mathit{hiding}\ c' & \Delta \vdash c' = c : \kappa \end{array} \right\} \\
[\![\mathit{unpack}\langle\alpha, x\rangle = e_1\ \mathit{in}\ e_2]\!] & = & \left\{ \begin{array}{l|l} \mathit{unpack}\langle\alpha, y\rangle = e_1' & e_1' \in [\![e_1]\!] \\ \quad \mathit{in}(\lambda x_\alpha{:}R\langle\alpha:\kappa\rangle.\lambda x{:}\alpha.e_2') & e_2' \in [\![e_2]\!] \\ \qquad (\pi_1 y)(\pi_2 y) & \end{array} \right\} \\
[\![\mathit{typerec}[\alpha.\sigma]\ c\ (e_{int}, e_\to, e_\times)]\!] & = & \left\{ \begin{array}{l|l} \mathit{typerec}[\alpha.\sigma']\ e & \sigma' \in [\![\sigma]\!] \\ \quad (e_{int}', e_\to', e_\times') & e \in \overline{\mathcal{R}[\![c]\!]}, e_{int}' \in [\![e_{int}]\!] \\ & e_\to' \in [\![e_\to]\!], e_\times' \in [\![e_\times]\!] \end{array} \right\}
\end{array}
$$

Follows the proof of Lemma 3.4.2, which states that $\Delta; R_{\mathsf{val}}(\Delta) \vdash \mathcal{R}|c| : R\langle c : \kappa\rangle$. □

We must next establish how substitution interacts with these operations. In the following, we will use the following abbreviations (where S_1 and S_2 are arbitrary sets of terms):

$$S_1[S_2/x] \stackrel{\text{def}}{=} \{e[e'/x] \mid e \in S_1 \;\&\; e' \in S_2\}$$
$$S_1[e'/x] \stackrel{\text{def}}{=} S_1[\{e'\}/x]$$

The following substitution lemmas will all be inclusions instead of equalities. The reason is that substitution can make more constructors equivalent to each other. Consider the following lemma:

Lemma 3.4.6 *For all $\Delta, \alpha{:}\kappa \vdash c' : \kappa'$ and $\Delta \vdash c : \kappa$, then $[\![c']\!][c/\alpha] \subseteq [\![c'[c/\alpha]]\!]$.*

Proof

This lemma is equivalent to showing that

$$\{c_1 \mid \Delta, \alpha{:}\kappa \vdash c_1 = c' : \kappa'\}[c/\alpha] \subseteq \{c_1 \mid \Delta \vdash c_1 = c'[c/\alpha] : \kappa'\}$$

This result directly follows from the substitution lemma for constructor equality. □

This is a strict inclusion as the substitution on the right side could introduce equalities that have no counterpart on the left side. For example, say $c = \lambda\beta.\beta$ and $c_2 = \alpha\ \mathit{int}$. The left set includes the constructor int (as $\vdash \mathit{int} = (\lambda\beta.\beta)\ \mathit{int} : \star$), but the right side does not, as int does not equal $\alpha\ \mathit{int}$ when α is abstract.

Lemma 3.4.7 (Substitution) *We must show a number of substitution properties:*

1. *If $\Delta, \alpha{:}\kappa \vdash c' : \kappa'$ and $\Delta \vdash c : \kappa$, then $\mathcal{R}[\![c']\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \subseteq \mathcal{R}[\![c'[c/\alpha]]\!]$.*
2. *If $\Delta, \alpha{:}\kappa \vdash c' : \kappa'$ and $\Delta \vdash c : \kappa$, then $\mathcal{R}[\![c']\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c'[c/\alpha]]\!]}$.*
3. *If $\Delta, \alpha{:}\kappa \vdash c' : \kappa'$ and $\Delta \vdash c : \kappa$, then $\overline{\mathcal{R}[\![c']\!]}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c'[c/\alpha]]\!]}$.*
4. *If $\Delta, \alpha{:}\kappa \vdash_i \sigma$ and $\Delta \vdash c : \kappa$ then $[\![\sigma]\!][c/\alpha] \subseteq [\![\sigma[c/\alpha]]\!]$.*
5. *If $\Delta, \alpha{:}\kappa; \Gamma \vdash_i e : \sigma$ and $\Delta \vdash c' = c : \kappa$, then $[\![e]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq [\![e[c/\alpha]]\!]$.*
6. *If $\Delta; \Gamma, x{:}\sigma \vdash_i e : \sigma'$ and $\Delta; \Gamma \vdash_i v : \sigma$ then $[\![e]\!][[\![v]\!]/x] = [\![e[v/x]]\!]$.*

Proof

By structural induction on c', σ and e.

1. Proof is by structural induction on c'.

case $c' \equiv \alpha$

$$\{x_\alpha\}[\mathcal{R}[\![c]\!]/x_\alpha] = \mathcal{R}[\![c]\!] = \mathcal{R}[\![\alpha[c/\alpha]]\!]$$

case $c' \equiv \beta$

$$\mathcal{R}[\![\beta]\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] = \{x_\beta\} = \mathcal{R}[\![\beta[c/\alpha]]\!]$$

case $c' \equiv \lambda\beta{:}\kappa.c''$

$$\begin{aligned}
&\mathcal{R}[\![\lambda\beta{:}\kappa.c'']\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \\
&\quad = \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle\beta : \kappa\rangle.e \mid e \in \mathcal{R}[\![c'']\!]\}[c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \\
&\quad \subseteq \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle\beta : \kappa\rangle.e \mid e \in \mathcal{R}[\![c''[c/\alpha]]\!]\} \\
&\quad = \mathcal{R}[\![\lambda\beta{:}\kappa.c''[c/\alpha]]\!]
\end{aligned}$$

as by induction $\mathcal{R}[\![c']\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \subseteq \mathcal{R}[\![c''[c/\alpha]]\!]$.

case $c' \equiv c_1 c_2$

$$\begin{aligned}
&\{e_1[c_2']e_2 \mid e_1 \in \mathcal{R}[\![c_1]\!], e_2 \in \mathcal{R}[\![c_2]\!], \Delta, \alpha{:}\kappa \vdash c_2 = c_2' : \kappa\}[c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \\
&\quad \subseteq \{e_1[c_2']e_2 \mid e_1 \in \mathcal{R}[\![c_1]\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha], e_2 \in \mathcal{R}[\![c_2]\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha], \\
&\qquad \Delta \vdash c_2[c/\alpha] = c_2' : \kappa\} \\
&\quad \subseteq \{e_1[c_2']e_2 \mid e_1 \in \mathcal{R}[\![c_1[c/\alpha]]\!], e_2 \in \mathcal{R}[\![c_2[c/\alpha]]\!], \Delta \vdash c_2[c/\alpha] = c_2' : \kappa\} \\
&\quad = \mathcal{R}[\![(c_1 c_2)[c/\alpha]]\!]
\end{aligned}$$

as by induction $\mathcal{R}[\![c_i]\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \subseteq \mathcal{R}[\![c_i[c/\alpha]]\!]$ for $i = 1, 2$.

case $c' \equiv \mathit{int}$

$$\mathcal{R}[\![\mathit{int}]\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] = \{R_{\mathit{int}}\} = \mathcal{R}[\![\mathit{int}[c/\alpha]]\!]$$

case $c' \equiv \rightarrow, \times$. Analogous to the previous.

case $c' \equiv \mathit{Typerec}[\kappa]\ \tau\ (c_{\mathit{int}}, c_{\rightarrow}, c_{\times})$.

Using the same definitions of $\mathit{rec}(\cdot)$ and $\mathit{expand}(\cdot)$ in Table 3.8, let

$$e \equiv \mathit{typerec}[\alpha.R\langle \mathit{rec}(\alpha) : \kappa\rangle]\ e_1\ (e'_{\mathit{int}}, \mathit{expand}(e'_{\rightarrow}), \mathit{expand}(e'_{\times}))$$

.

Then $\mathcal{R}[\![c']\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha]$

$$\begin{aligned}
&= \{e \mid e_{\mathit{int}} \in \mathcal{R}[\![c_{\mathit{int}}]\!], e_{\rightarrow} \in \mathcal{R}[\![c_{\rightarrow}]\!], e_{\times} \in \mathcal{R}[\![c_{\times}]\!], e_1 \in \mathcal{R}[\![\tau]\!]\ \} \\
&\quad [c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \\
&\subseteq \{e \mid e_{\mathit{int}} \in \mathcal{R}[\![c_{\mathit{int}}[c/\alpha]]\!], e_{\rightarrow} \in \mathcal{R}[\![c_{\rightarrow}[c/\alpha]]\!], \\
&\quad e_{\times} \in \mathcal{R}[\![c_{\times}[c/\alpha]]\!], e_1 \in \mathcal{R}[\![\tau[c/\alpha]]\!]\ \} \\
&= \mathcal{R}[\![c'[c/a]]\!]
\end{aligned}$$

as by induction, $\mathcal{R}[\![c_i]\!][c/\alpha][\mathcal{R}[\![c]\!]/x_\alpha] \subseteq \mathcal{R}[\![c_i[c/\alpha]]\!]$,for $i = 1, \mathit{int}, \rightarrow, \times$, and the result from lemma 3.4.6.

2. Proof is by structural induction on c'.

case $c' \equiv \alpha$

$$\mathcal{R}[\![\alpha]\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] = \{x_\alpha\}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] = \overline{\mathcal{R}[\![c]\!]} = \overline{\mathcal{R}[\![\alpha[c/\alpha]]\!]}$$

case $c' \equiv \beta$

$$\mathcal{R}[\![\beta]\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] = \{x_\beta\}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] = \{x_\beta\} \subseteq \overline{\mathcal{R}[\![\beta]\!]}$$

case $c' \equiv \lambda\beta{:}\kappa.c''$

$$\begin{aligned}
&\mathcal{R}[\![\lambda\beta{:}\kappa.c'']\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \\
&\quad = \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle\beta:\kappa\rangle.e \mid e \in \overline{\mathcal{R}[\![c'']\!]}\}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \\
&\quad \subseteq \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle\beta:\kappa\rangle.e \mid e \in \overline{\mathcal{R}[\![c''[c/\alpha]]\!]}\} \\
&\quad \subseteq \overline{\mathcal{R}[\![\lambda\beta{:}\kappa.c''[c/\alpha]]\!]}
\end{aligned}$$

as by induction $\mathcal{R}[\![c'']\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c''[c/\alpha]]\!]}$

case $c' \equiv c_1 c_2$

$$\begin{aligned}
\mathcal{R}[\![c_1c_2]\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] &= \{e_1[c_2']e_2 \mid e_i \in \mathcal{R}[\![c_i]\!], \\
&\qquad\quad \Delta \vdash c_2 = c_2' : \kappa\}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \\
&\subseteq \{e_1[c_2'']e_2 \mid e_i \in \overline{\mathcal{R}[\![c_i[c/\alpha]]\!]}, \\
&\qquad\quad \Delta \vdash c_2[c/\alpha] = c_2'' : \kappa\} \\
&\subseteq \overline{\mathcal{R}[\![(c_1c_2)[c/\alpha]]\!]}
\end{aligned}$$

as by induction $\mathcal{R}[\![c_i]\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c_i[c/\alpha]]\!]}$

case $c' \equiv int$

$$\mathcal{R}[\![int]\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] = \{R_{int}\} \subseteq \overline{\mathcal{R}[\![int[c/\alpha]]\!]}$$

case $c' \equiv \to, \times$. Analogous to the previous.

case $c' \equiv Typerec[\kappa]\ c\ (c_{int}, c_{\to}, c_{\times})$

Again, using the same definitions of $rec(\cdot)$ and $expand(\cdot)$ in Table 3.8, let

$$e \equiv typerec[\alpha.R\langle rec(\alpha) : \kappa\rangle]\ e_1\ (e'_{int}, expand(e'_{\to}), expand(e'_{\times}))$$

. Then $\mathcal{R}[\![c']\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha]$

$$\begin{aligned}&= \{e \mid e_{int} \in \mathcal{R}[\![c_{int}]\!], e_{\rightarrow} \in \mathcal{R}[\![c_{\rightarrow}]\!], e_{\times} \in \mathcal{R}[\![c_{\times}]\!], e_1 \in \mathcal{R}[\![\tau]\!]\ \} \\ &\quad [c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \\ &\subseteq \{e \mid e_{int} \in \overline{\mathcal{R}[\![c_{int}[c/\alpha]]\!]}, \\ &\quad e_{\rightarrow} \in \overline{\mathcal{R}[\![c_{\rightarrow}[c/\alpha]]\!]}, e_{\times} \in \overline{\mathcal{R}[\![c_{\times}[c/\alpha]]\!]}, e_1 \in \overline{\mathcal{R}[\![\tau[c/\alpha]]\!]}\ \} \\ &= \overline{\mathcal{R}[\![c'[c/a]]\!]}\end{aligned}$$

as by induction, $\mathcal{R}[\![c_i]\!][c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c_i[c/\alpha]]\!]}$,for $i = 1, int, \rightarrow, \times$.

3. Corollary of 2. Say $\Delta, \alpha{:}\kappa \vdash c' : \kappa'$, $\Delta \vdash c : \kappa$, and we wish to show that

$$\overline{\mathcal{R}[\![c']\!]}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c'[c/\alpha]]\!]}.$$

Let $e \in \overline{\mathcal{R}[\![c']\!]}[c/\alpha][\overline{\mathcal{R}[\![c]\!]}/x_\alpha]$ be arbitrary. The e is of the form

$$\mathcal{R}[\![c'']\!][c_1/\alpha][\mathcal{R}[\![c_2]\!]/x_\alpha],$$

where $\Delta \vdash c'' = c' : \kappa'$ and (by abuse of notation) $\Delta \vdash c_1 = c_2 = c : \kappa$. By part 2, e is in $\overline{\mathcal{R}[\![c''[c_1/\alpha]]\!]}$, which is equal to $\overline{\mathcal{R}[\![c'[c/\alpha]]\!]}$.

4. Proof by induction on σ.

case $\sigma \equiv T(c'')$

$$\begin{aligned}[\![\sigma]\!][c/\alpha] &= \{T(c') \mid \Delta, \alpha{:}\kappa \vdash c' = c'' : \star\}[c/\alpha] \\ &\subseteq \{T(c') \mid \Delta \vdash c' = c''[c/\alpha] : \star\} \\ &= [\![\sigma[c/\alpha]]\!]\end{aligned}$$

case $\sigma \equiv int$

$$[\![int]\!][c/\alpha] = \{int\} = [\![int[c/\alpha]]\!]$$

case $\sigma \equiv \sigma_1 \rightarrow \sigma_2$

$$\begin{aligned}&[\![\sigma_1 \rightarrow \sigma_2]\!][c/\alpha] \\ &\quad = \{\sigma_1' \rightarrow \sigma_2' \mid \sigma_i' \in [\![\sigma_i]\!],\ i = i, 1\}[c/\alpha] \\ &\quad \subseteq \{\sigma_1' \rightarrow \sigma_2' \mid \sigma_i' \in [\![\sigma_i[c/a]]\!],\ i = i, 1\} \\ &\quad = [\![(\sigma_1 \rightarrow \sigma_2)[c/a]]\!]\end{aligned}$$

as by induction, $[\![\sigma_i]\!][c/a] \subseteq [\![\sigma_i[c/a]]\!]$.

case $\sigma \equiv \sigma_1 \times \sigma_2$ Analogous to the previous case.

case $\sigma \equiv \forall\alpha{:}\kappa.\sigma$

$$\begin{aligned}&[\![\forall\alpha{:}\kappa.\sigma]\!][c/\alpha]\\ &\quad= \{\forall\alpha{:}\kappa.\sigma' \mid \sigma' \in [\![\sigma]\!]\}[c/\alpha]\\ &\quad\subseteq \{\forall\alpha{:}\kappa.\sigma' \mid \sigma'_i \in [\![\sigma_i[c/a]]\!]\}\\ &\quad= [\![(\forall\alpha{:}\kappa.\sigma)[c/a]]\!]\end{aligned}$$

as by induction, $[\![\sigma]\!][c/a] \subseteq [\![\sigma[c/a]]\!]$.

case $\sigma \equiv \exists\alpha{:}\kappa.\sigma$ Analogous to the previous case.

5. Proof by induction on e.

case $e \equiv i$ Trivial.

case $e \equiv x$ Trivial.

case $e \equiv \lambda x{:}\sigma.e'$

$$\begin{aligned}&[\![\lambda x{:}\sigma.e']\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\ &\quad= \{\lambda x{:}\sigma'.e'' \mid \sigma' \in [\![\sigma]\!][c'/\alpha], e'' \in [\![e']\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\}\\ &\quad\subseteq \{\lambda x{:}\sigma'.e'' \mid \sigma' \in [\![\sigma[c'/\alpha]]\!], e'' \in [\![e'[c'/\alpha]]\!]\}\\ &\quad= [\![(\lambda x{:}\sigma.e')[c'/\alpha]]\!]\end{aligned}$$

By induction $[\![e']\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq [\![e'[c'/\alpha]]\!]$ and by lemma $[\![\sigma]\!][c'/\alpha] \subseteq [\![\sigma[c'\alpha]]\!]$.

case $e \equiv \mathit{fix}\, x{:}\sigma.e'$ Analogous to previous case.

case $e \equiv e_1 e_2$

$$\begin{aligned}&[\![e_1 e_2]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\ &\quad= \{e'_1 e'_2 \mid e'_1 \in [\![e_1]\!], e'_2 \in [\![e_2]\!]\}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\ &\quad\subseteq \{e'_1 e'_2 \mid e'_1 \in [\![e_1[c'/\alpha]]\!], e'_2 \in [\![e_2[c'/\alpha]]\!]\}\\ &\quad= [\![e_1 e_2[c'/\alpha]]\!]\end{aligned}$$

By induction $[\![e'_i]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq [\![e'_i[c'/\alpha]]\!]$.

case $e \equiv \langle e_1, e_2\rangle$ Analogous to the previous case.

case $e \equiv \pi_i e'$ Analogous to the previous case.

case $e \equiv \Lambda\beta{:}\kappa.e'$ Follows by induction:

$$\begin{aligned}&[\![\Lambda\beta{:}\kappa.e']\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\ &\quad= \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle\beta : \kappa\rangle.e'' \mid e'' \in [\![e']\!]\}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\ &\quad\subseteq \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle\beta : \kappa\rangle.e'' \mid e'' \in [\![e'[c'/\alpha]]\!]\}\\ &\quad= [\![\Lambda\beta{:}\kappa.e'[c'/\alpha]]\!]\end{aligned}$$

case $e \equiv e'[c_1]$.

$$\begin{aligned}&[\![e'[c_1]]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad= \{e_1[c_2]e_2 \mid e_1 \in [\![e']\!], \Delta, \alpha{:}\kappa' \vdash c_1 = c_2 : \kappa, e_2 \in \overline{\mathcal{R}[\![c_2]\!]}\}\\&\qquad[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad\subseteq [\![(e'[c_1])[c'/\alpha]]\!]\end{aligned}$$

By induction, $[\![e']\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq [\![e[c/\alpha]]\!]$. By the previous part 3,

$$\overline{\mathcal{R}[\![c_2]\!]}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c_2[c/\alpha]]\!]}.$$

case $e \equiv \mathit{pack}\, e'\ \mathit{as}\ \exists\beta{:}\kappa.\sigma\ \mathit{hiding}\ c_1$

$$\begin{aligned}&[\![\mathit{pack}\ \ e\ \mathit{as}\ \exists\beta{:}\kappa.\sigma\ \mathit{hiding}\ c_1]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad= \{\mathit{pack}\langle e_c, e'\rangle\ \mathit{as}\ \exists\beta{:}\kappa.R\langle\beta:\kappa\rangle \times \sigma'\ \mathit{hiding}\ c_1'\\&\qquad\mid e_c \in \overline{\mathcal{R}[\![c_1]\!]}, c_1' \in [\![c_1]\!], e' \in [\![e]\!], \sigma' \in [\![\sigma]\!]\}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad\subseteq \{\mathit{pack}\langle e_c, e'\rangle\ \mathit{as}\ \exists\beta{:}\kappa.R\langle\beta:\kappa\rangle \times \sigma'\ \mathit{hiding}\ c_1'\\&\qquad\mid e_c \in \overline{\mathcal{R}[\![c_1[c'/\alpha]]\!]}, c_1' \in [\![c_1[c'/\alpha]]\!], e' \in [\![e[c'/\alpha]]\!], \sigma' \in [\![\sigma[c'/\alpha]]\!]\}\\&\quad= [\![(\mathit{pack}\ \ e\ \mathit{as}\ \exists\beta{:}\kappa.\sigma\ \mathit{hiding}\ c_1)[c'/\alpha]]\!]\end{aligned}$$

as by part 3, $\overline{\mathcal{R}[\![c_1]\!]}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq \overline{\mathcal{R}[\![c_1[c'/\alpha]]\!]}$, by lemma 3.4.6

$$\{c_1' \mid \Delta \vdash c_1 = c_1' : \kappa\}[c'/\alpha] \subseteq \{c_1' \mid \Delta \vdash c_1[c'/\alpha] = c_1' : \kappa\}$$

and by induction $[\![e']\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha] \subseteq [\![e'[c'/\alpha]]\!]$.

case $e \equiv \mathit{unpack}\langle\beta, x\rangle = e_1\ \mathit{in}\ e_2$ Follows directly by induction.

$$\begin{aligned}&[\![e]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad= \{\mathit{unpack}\langle\beta, y\rangle = e_1'\ \mathit{in}(\lambda x_\beta{:}R\langle\beta:\kappa\rangle.\lambda x:\beta.e_2')(\pi_1 y)(\pi_2 y)\\&\qquad\mid e_i' \in [\![e_i]\!]\}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad\subseteq \{\mathit{unpack}\langle\beta, y\rangle = e_1'\ \mathit{in}(\lambda x_\beta{:}R\langle\beta:\kappa\rangle.\lambda x:\beta.e_2')(\pi_1 y)(\pi_2 y)\\&\qquad\mid e_i' \in [\![e_i[c'/\alpha]]\!]\}\\&\quad= [\![e[c'/\alpha]]\!]\end{aligned}$$

case $e \equiv \mathit{typerec}[\alpha.\sigma]\ c''\ (e_{\mathit{int}}, e_{\to}, e_{\times})$

$$\begin{aligned}&[\![e]\!][c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad= \{\mathit{typerec}[\alpha.\sigma']\ e'\ (e_{\mathit{int}}{}', e_{\to}{}', e_{\times}{}')\\&\qquad\mid \sigma' \in [\![\sigma]\!], e' \in \overline{\mathcal{R}[\![c'']\!]}, e_i' \in [\![e_i]\!]\ \text{ for } i = \mathit{int}, \to, \times\}[c'/\alpha][\overline{\mathcal{R}[\![c']\!]}/x_\alpha]\\&\quad\subseteq \{\mathit{typerec}[\alpha.\sigma']\ e'\ (e_{\mathit{int}}{}', e_{\to}{}', e_{\times}{}')\\&\qquad\mid \sigma' \in [\![\sigma[c'/\alpha]]\!], e' \in \overline{\mathcal{R}[\![c''[c'/\alpha]]\!]}, e_i' \in [\![e_i[c'/\alpha]]\!]\ \text{for } i = \mathit{int}, \to, \times\}\\&\quad= [\![e[c'/\alpha]]\!]\end{aligned}$$

6. Proof is by induction on e.

case $e \equiv i$ Trivial.

case $e \equiv x$ Trivial.

case $e \equiv \lambda y{:}\sigma.e'$.

$$\begin{aligned}\llbracket \lambda y{:}\sigma.e' \rrbracket[\llbracket v \rrbracket/x] &= \{\lambda y{:}\sigma'.e''[v'/x] \mid \sigma' \in \llbracket \sigma \rrbracket, e'' \in \llbracket e' \rrbracket, v' \in \llbracket v \rrbracket\} \\ &= \{\lambda y{:}\sigma'.e'' \mid \sigma' \in \llbracket \sigma \rrbracket, e'' \in \llbracket e'[v/x] \rrbracket\} \\ &= \llbracket (\lambda y{:}\sigma.e)[v/x] \rrbracket\end{aligned}$$

as by induction $\llbracket e' \rrbracket[\llbracket v \rrbracket/x] = \llbracket e'[v/x] \rrbracket$.

case $e \equiv \mathit{fix}\, x{:}\sigma.e'$ Analogous to the previous case.

case $e \equiv e_1 e_2$

$$\begin{aligned}\llbracket e_1 e_2 \rrbracket[\llbracket v \rrbracket/x] &= \{e_1' e_2' \mid e_1' \in \llbracket e_1 \rrbracket, e_1' \in \llbracket e_1 \rrbracket\}[\llbracket v \rrbracket/x] \\ &= \{e_1' e_2' \mid e_1' \in \llbracket e_1[\llbracket v \rrbracket/x] \rrbracket, e_1' \in \llbracket e_1[\llbracket v \rrbracket/x] \rrbracket\} \\ &= \llbracket (e_1 e_2)[v/x] \rrbracket\end{aligned}$$

as by induction $\llbracket e_i \rrbracket[\llbracket v \rrbracket/x] = \llbracket e_i[v/x] \rrbracket$.

case $e \equiv \langle e_1, e_2 \rangle$ Analogous to the previous case.

case $e \equiv \pi_i e'$ Analogous to the previous case.

case $e \equiv \Lambda\alpha{:}\kappa.e'$

$$\begin{aligned}\llbracket \Lambda\alpha{:}\kappa.e' \rrbracket[\llbracket v \rrbracket/x] &= \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle \beta : \kappa \rangle).e'' \mid e'' \in \llbracket e' \rrbracket\}[\llbracket v \rrbracket/x] \\ &= \{\Lambda\beta{:}\kappa.\lambda x_\beta{:}R\langle \beta : \kappa \rangle).e'' \mid e'' \in \llbracket e'[\llbracket v \rrbracket/x] \rrbracket\} \\ &= \llbracket (\Lambda\beta{:}\kappa.e')[c'/\alpha] \rrbracket\end{aligned}$$

as by induction $\llbracket e' \rrbracket[\llbracket v \rrbracket/x] = \llbracket e'[v/x] \rrbracket$.

case $e \equiv e'[c_1]$.

$$\begin{aligned}&\llbracket e'[c_1] \rrbracket[\llbracket v \rrbracket/x] \\ &\quad = \{e_1[c_2]e_2 \mid e_1 \in \llbracket e' \rrbracket, \Delta, \alpha{:}\kappa' \vdash c_1 = c_2 : \kappa, e_2 \in \overline{\mathcal{R}\llbracket c_2 \rrbracket}\}[\llbracket v \rrbracket/x] \\ &\quad \subseteq \llbracket (e'[c_1])[v/x] \rrbracket\end{aligned}$$

as by induction, $\llbracket e' \rrbracket[\llbracket v \rrbracket/x] = \llbracket e[\llbracket v \rrbracket/x] \rrbracket$.

case $e \equiv \mathit{pack}\, e'\, \mathit{as}\, \exists\beta{:}\kappa.\sigma\, \mathit{hiding}\, c$ Follows by induction.

$$\begin{aligned}\llbracket e \rrbracket[\llbracket v \rrbracket/x] &= \{\mathit{pack}\langle e_c, e' \rangle\, \mathit{as}\, \exists\beta{:}\kappa.R\langle \beta : \kappa \rangle \times \sigma'\, \mathit{hiding}\, c_1' \\ &\quad \mid e_c \in \overline{\mathcal{R}\llbracket c_1 \rrbracket}, \Delta \vdash c_1 = c_1' : \kappa, e' \in \llbracket e \rrbracket, \sigma' \in \llbracket \sigma \rrbracket\}[\llbracket v \rrbracket/x] \\ &= \{\mathit{pack}\langle e_c, e' \rangle\, \mathit{as}\, \exists\beta{:}\kappa.R\langle \beta : \kappa \rangle \times \sigma'\, \mathit{hiding}\, c_1' \\ &\quad \mid e_c \in \overline{\mathcal{R}\llbracket c_1 \rrbracket}, \Delta \vdash c_1 = c_1' : \kappa, e' \in \llbracket e[v/\alpha] \rrbracket, \sigma' \in \llbracket \sigma \rrbracket\} \\ &= \llbracket (\mathit{pack}\ \ e\, \mathit{as}\, \exists\beta{:}\kappa.\sigma\, \mathit{hiding}\, c_1)[v/x] \rrbracket\end{aligned}$$

case $e \equiv unpack\langle\beta, x\rangle = e_1 \, in \, e_2$ Follows by induction.

$$\begin{aligned}
&[\![e]\!][[\![v]\!]/x] \\
&\quad = \{unpack\langle\beta, x\rangle = e_1' \; in (\lambda x_\beta{:}R\langle\beta : \kappa\rangle.\lambda x : \beta.e_2')(\pi_1 y)(\pi_2 y) \\
&\quad\quad \mid e_i' \in [\![e_i]\!]\}[[\![v]\!]/x] \\
&\quad = \{unpack\langle\beta, x\rangle = e_1' \; in (\lambda x_\beta{:}R\langle\beta : \kappa\rangle.\lambda x : \beta.e_2')(\pi_1 y)(\pi_2 y) \\
&\quad\quad \mid e_i' \in [\![e_i[v/x]]\!]\} \\
&\quad = [\![e[v/x]]\!]
\end{aligned}$$

case $e \equiv typerec[\alpha.\sigma] \; c'' \; (e_{int}, e_{\to}, e_{\times})$ Follows by induction.

$$\begin{aligned}
[\![e]\!][[\![v]\!]/x] &= \{typerec[\alpha.\sigma'] \; \underline{e'} \; (e_{int}{}', e_{\to}{}', e_{\times}{}') \\
&\quad \mid \sigma' \in [\![\sigma]\!], e' \in \overline{\mathcal{R}[\![c'']\!]}, e_i' \in [\![e_i]\!] \;\; \text{for } i = int, \to, \times\}[[\![v]\!]/x] \\
&= \{typerec[\alpha.\sigma'] \; \underline{e'} \; (e_{int}{}', e_{\to}{}', e_{\times}{}') \\
&\quad \mid \sigma' \in [\![\sigma]\!], e' \in \overline{\mathcal{R}[\![c'']\!]}, e_i' \in [\![e_i[[\![v]\!]/x]]\!] \text{ for } i = int, \to, \times\} \\
&= [\![e[v/x]]\!]
\end{aligned}$$

□

Next, we also need to establish that the evaluation of term representations agrees with constructor equality. In the end, our goal is to show that if $e \in \overline{\mathcal{R}[\![int]\!]}$ then e must evaluate to R_{int} (and similar results for arrow and product types).

Lemma 3.4.8 *If $v \in \mathcal{R}[\![c]\!]$ and $\emptyset \vdash c : \star$ then c is in normal form.*

Proof

Proof by induction on c.

case $c \equiv int$, in normal form.

case $c \equiv c_1 \to c_2$. By induction c_1 and c_2 are normal, so $c_1 \to c_2$ is normal.

case $c \equiv c_1 \times c_2$ By induction c_1 and c_2 are normal, so $c_1 \times c_2$ is normal.

case $c \not\equiv \alpha, (\lambda\alpha{:}\kappa.c), (c_1 c_2)$, or $Typerec \;\; c \;\; (c_{int}, c_{\to}, c_{\times})$ because either c is not closed or of the right kind (in the former two cases) or $\mathcal{R}[\![c]\!]$ is not a value (latter two cases).

□

Lemma 3.4.9 *For all $\emptyset \vdash c : \kappa$, $e \in \mathcal{R}[\![c]\!]$ then either e is a value or there exists some e' and c' such that $e \mapsto^+ e'$ and $e' \in \mathcal{R}[\![c']\!]$ and c reduces to c'.*

Proof

Proof is by induction on $\emptyset \vdash c : \kappa$.

case $\vdash int : \star$.

In this case $\mathcal{R}[\![int]\!]$ is only R_{int}, a value.

case $\vdash \rightarrow : \star \rightarrow \star \rightarrow \star$

case $\vdash \times : \star \rightarrow \star \rightarrow \star$. Same as above.

case Variable case cannot occur in closed constructors.

case

$$[cfn]\quad \frac{\emptyset, \alpha{:}\kappa_1 \vdash c : \kappa_2}{\alpha{:}\kappa_1 \vdash \lambda\alpha{:}\kappa_1.c : \kappa_1 \rightarrow \kappa_2}$$

In this case $\mathcal{R}[\![\lambda\alpha{:}\kappa_1.c]\!]$ is a value.

case

$$[capp]\quad \frac{\emptyset \vdash c_1 : \kappa_1 \rightarrow \kappa_2 \qquad \emptyset \vdash c_2 : \kappa_1}{\emptyset \vdash c_1 c_2 : \kappa_2}$$

In this case, $\mathcal{R}[\![c_1 c_2]\!] = \{e_1[c_2']e_2 \mid e_i \in \mathcal{R}[\![c_i]\!], \emptyset \vdash c_2 = c_2' : \kappa_1\}$ There are three cases to consider:

- e_1 and e_2 are values. As $e_1 \in \mathcal{R}[\![c_1]\!]$, then $\emptyset \vdash e_1 : R\langle c_1 : \kappa_1 \rightarrow \kappa_2\rangle$. By canonical forms, e_1 must be $\Lambda\alpha{:}\kappa_1.\lambda x{:}R\langle a : \kappa_1\rangle.e_1'$. Furthermore, c_1 must be of the form $\lambda\alpha{:}\kappa_1.c_1'$ where $e_1' \in \mathcal{R}[\![c_1']\!]$ as this is the only case of $\mathcal{R}[\![\cdot]\!]$ that produces a term of this form. Therefore $e_1[c_2']e_2 \mapsto e_1'[c_2'/\alpha][e_2/x_\alpha]$. By substitution corollary 3 this term is in $\mathcal{R}[\![c_1[c_2/\alpha]]\!]$. As $(\lambda\alpha{:}\kappa_1.c_1')c_2$ reduces to $c_1[c_2/\alpha]$ we are done.
- $e_1 \mapsto e_1'$. Then $e_1[c_2']e_2 \mapsto e_1'[c_2']e_2$. By induction, there exists a c_1' such that e_1' is in $\mathcal{R}[\![c_1']\!]$ and c_1 reduces to c_1'. Therefore, $e_1'[c_2']e_2 \in \mathcal{R}[\![c_1' c_2]\!]$ and $c_1 c_2$ reduces to $c_1' c_2$.
- e_1 is a value and $e_2 \mapsto e_2'$. Then $e_1[c_2']e_2 \mapsto e_1[c_2']e_2'$. This case is analogous to the previous.

case

$$[ctrec]\quad \frac{\begin{array}{c}\emptyset \vdash \tau : \star \qquad \emptyset \vdash c_{int} : \kappa \\ \emptyset \vdash c_{\rightarrow} : \star \rightarrow \star \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\ \emptyset \vdash c_{\times} : \star \rightarrow \star \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa\end{array}}{\emptyset \vdash \mathit{Typerec}\ \tau\,(c_{int}, c_{\rightarrow}, c_{\times}) : \kappa}$$

In this case $\mathcal{R}[\![c]\!]$ is

$$typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ e\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times}))$$

where

$$e_{int} \in \mathcal{R}[\![c_{int}]\!], e_{\rightarrow} \in \mathcal{R}[\![c_{\rightarrow}]\!], e_{\times} \in \mathcal{R}[\![c_{\times}]\!], e \in \mathcal{R}[\![\tau]\!]$$

Suppose $e \in \mathcal{R}[\![\tau]\!]$ is a value. By lemma 3.4.8, τ must be either int, $c_1 \rightarrow c_2$ or $c_1 \times c_2$.

- If $\tau \equiv int$, then e is R_{int}. $\mathcal{R}[\![c]\!] \mapsto e_{int} \in \mathcal{R}[\![c_{int}]\!]$.
  As $Typerec\ \ int\ (c_{int}, c_{\rightarrow}, c_{\times})$ reduces to c_{int}, the result holds.
- If $\tau \equiv c_1 \rightarrow c_2$, then by definition of $\mathcal{R}[\![\tau]\!]$, e is $R_{\rightarrow}[c'_1]v_1[c'_2]v_2$, where $c'_i \in [\![c_i]\!]$ and $v_i \in \mathcal{R}[\![c_i]\!]$. Therefore
  $$\begin{aligned}
  \mathcal{R}[\![c]\!] \mapsto\ & expand(e_{\rightarrow}) \\
  & \quad [c'_1]\ v_1 \\
  & \quad\quad (typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ v_1\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times}))) \\
  & \quad [c'_2]\ v_2 \\
  & \quad\quad (typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ v_2\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times}))) \\
  \mapsto^6\ & e_{\rightarrow} \\
  & \quad [c'_1]\ v_1\ [rec(c'_1)] \\
  & \quad\quad (typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ v_1\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times}))) \\
  & \quad [c'_2]\ v_2\ [rec(c'_1)] \\
  & \quad\quad (typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ v_2\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times}))) \\
  \in\ & \mathcal{R}[\![c_{\rightarrow}\, c_1 (Typerec\ c_1(c_{int}, c_{\rightarrow}, c_{\times}))\ c_2\ (Typerec\ c_2(c_{int}, c_{\rightarrow}, c_{\times}))]\!]
  \end{aligned}$$
  As $Typerec\ (c_1 \rightarrow c_2)\ (c_{int}, c_{\rightarrow}, c_{\times})$ reduces to
  $$c_{\rightarrow}\ \ c_1\ c_2\ (Typerec\ \ c_1(c_{int}, c_{\rightarrow}, c_{\times}))\ (Typerec\ \ c_2(c_{int}, c_{\rightarrow}, c_{\times})),$$
  this evaluation is correct.
- If $c \equiv c_1 \times c_2$, the case is analogous to the previous.

If e is not a value, then by induction it steps to $e' \in \overline{\mathcal{R}[\![\tau]\!]}$, and

$$\begin{aligned}
& typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ e\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times})) \\
& \quad \mapsto typerec[\alpha.R\langle rec(\alpha):\kappa\rangle]\ e'\ (e_{int}, expand(e_{\rightarrow}), expand(e_{\times})) \\
& \quad \in \overline{\mathcal{R}[\![c]\!]}
\end{aligned}$$

□

Lemma 3.4.10 *If $e \in \overline{\mathcal{R}[\![c]\!]}$ then e evaluates to a value $v \in \overline{\mathcal{R}[\![c]\!]}$.*

Proof

If $e \in \overline{\mathcal{R}[\![c]\!]}$ then $e \in \mathcal{R}[\![c']\!]$ for some c' equal to c. Assume e diverges. By above, the evaluation sequence of e is $e = e_1 \mapsto^+ e_2 \mapsto^+ e_3 \ldots$, where $e_i \in \mathcal{R}[\![c_i]\!]$ and c_{i+1} reduces to c_i. If this sequence is infinite, there must be an infinite reduction sequence for c', which is impossible. Therefore, this sequence must be $e = e_1 \mapsto^+ e_2 \mapsto^+ e_3 \ldots \mapsto^+ e_n$, where $e_n \in \mathcal{R}[\![c_n]\!]$, for some c_n equal to c. Furthermore, e_n must be a value by type soundness. □

Corollary 3.4.11 *1. If $e \in \overline{\mathcal{R}[\![int]\!]}$ then e evaluates to R_{int}.*

2. If $e \in \overline{\mathcal{R}[\![\underline{\tau_1 \to \tau_2}]\!]}$ then e evaluates to $R_{\to}[\tau_1'] \; v_1 \; [\tau_2'] \; v_2$, where $\emptyset \vdash \tau_i = \tau_i' : \star$ and $v_i \in \overline{\mathcal{R}[\![\tau_i]\!]}$ for $i = 1, 2$.

3. If $e \in \overline{\mathcal{R}[\![\underline{\tau_1 \times \tau_2}]\!]}$ then e evaluates to $R_{\times}[\tau_1'] \; v_1 \; [\tau_2'] \; v_2$, where $\emptyset \vdash \tau_i = \tau_i' : \star$ and $v_i \in \overline{\mathcal{R}[\![\tau_i]\!]}$ for $i = 1, 2$.

Proof

1. By lemma 3.4.10, e evaluates to some $v \in \mathcal{R}[\![c]\!]$, where $\emptyset \vdash int = c : \star$. By lemma 3.4.8, c is in normal form, so it must be int. Therefore, v is R_{int}.

2. By lemma 3.4.10, e evaluates to some $v \in \mathcal{R}[\![c]\!]$, where $\emptyset \vdash c_1 \to c_2 = c : \star$. By lemma 3.4.8, c is in normal form, so it must be $c_1'' \to c_2''$ for some $\emptyset \vdash c_i = c_i' : \star$. Therefore v is $R_{\to}[c_1'] \; v_1 \; [c_2'] \; v_2$, where $\emptyset \vdash c_i = c_i' : \star$ and $v_i \in \mathcal{R}[\![c_i'']\!] = \overline{\mathcal{R}[\![c_i]\!]}$.

3. Analogous to the previous case.

□

Lemma 3.4.12 $[\![v]\!]$ *only contains values.*

Proof

Proof is by induction on v. □

Lemma 3.4.13 (Simulation) *If $\vdash_i e_1 : \sigma$ and $e_1 \mapsto_i e_2$ then for all $e_1' \in [\![e_1]\!]$ there exists an $e_2' \in [\![e_2]\!]$ such that $e_1 \mapsto_R^* e_2'$.*

Proof

Proof by induction on $e_1 \mapsto_i e_2$.

case $(\Lambda\alpha{:}\kappa.e)[c] \mapsto_i e[c/\alpha]$

In this case, $[\![e_1]\!] = (\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.e')[c']v$ where $e' \in [\![e]\!]$ and $\vdash c = c' : \kappa$ and $v \in \overline{\mathcal{R}[\![c]\!]}$ are arbitrary. This term steps to $e'[c'/\alpha][v/x_\alpha]$. By Lemma 5, $e'[c'/\alpha][v/x_\alpha] \in [\![e[c/\alpha]]\!]$.

case $(\mathit{fix}\ f{:}\sigma.v)[c] \mapsto_i (v[\mathit{fix}\ f{:}\sigma.v/f])[c]$

Here, $[\![e_1]\!] = (\mathit{fix}\ \ f{:}\sigma'.v')[c']e$ where $\emptyset \vdash \sigma = \sigma'$, $\emptyset \vdash c = c' : \kappa$ and $e \in \overline{\mathcal{R}[\![c]\!]}$. This term steps in LIR to $(v'[\mathit{fix}\ f{:}\sigma'.v'/f])[c']e$, which is in $[\![e_2]\!]$, by Lemma 6.

case Type analysis of *int*:

$$\frac{c \text{ normalizes to } \mathit{int}}{\mathit{typerec}\ \ c\ (e_{\mathit{int}}, e_{\to}, e_{\times}) \mapsto_i e_{\mathit{int}}}$$

Now $[\![e_1]\!] = \mathit{typerec}\, e\ (e'_{\mathit{int}}, e'_{\to}, e'_{\times})$, where $e \in \overline{\mathcal{R}[\![c]\!]}$, $e'_{\mathit{int}} \in [\![e_{\mathit{int}}]\!]$, and $e'_{\to} \in [\![e_{\to}]\!]$, $e'_{\times} \in [\![e_{\times}]\!]$. By lemma 3.4.11.1, e evaluates to R_{int}, so the term $[\![e_1]\!]$ steps to e'_{int}.

case Type analysis of an arrow type.

$$\frac{c \text{ normalizes to } (c_1 \to c_2)}{\begin{array}{l}\mathit{typerec}\ \ c\ (e_{\mathit{int}}, e_{\to}, e_{\times}) \\ \quad \mapsto_i e_{\to}[c_1](\mathit{typerec}\ \ c_1(e_{\mathit{int}}, e_{\to}, e_{\times}))\ [c_2]\ (\mathit{typerec}\ \ c_1(e_{\mathit{int}}, e_{\to}, e_{\times}))\end{array}}$$

As above, $[\![e_1]\!] = \mathit{typerec}\, e(e'_{\mathit{int}}, e'_{\to}, e'_{\times})$, where $e \in \overline{\mathcal{R}[\![c]\!]}$, $e'_{\mathit{int}} \in [\![e_{\mathit{int}}]\!]$, and $e'_{\to} \in [\![e_{\to}]\!]$, $e'_{\times} \in [\![e_{\times}]\!]$. However, this time, by lemma 3.4.11.2, e evaluates to $R_{\to}[c'_1]\ v_1\ [c'_2]v_2$, where $c'_1 \in [\![c_1]\!], c'_2 \in [\![c_2]\!]$, $v_1 \in \overline{\mathcal{R}[\![c_1]\!]}$ and $v_2 \in \overline{\mathcal{R}[\![c_2]\!]}$. Therefore, the term steps to

$$e'_{\to}[c'_1]\ v_1\ (\mathit{typerec}\ \ v_1\ (e_{\mathit{int}}, e_{\to}, e_{\times}))[c'_2]\ v_2\ (\mathit{typerec}\ \ v_2\ (e_{\mathit{int}}, e_{\to}, e_{\times}))$$

This result is in

$$[\![e_{\to}\ [c_1]\ (\mathit{typerec}\ \ c_1(e_{\mathit{int}}, e_{\to}, e_{\times}))\ [c_2]\ (\mathit{typerec}\ \ c_1(e_{\mathit{int}}, e_{\to}, e_{\times}))]\!].$$

case Type analysis of a product type. This case is analogous to the previous.

case $(\lambda x{:}\sigma.e)v \mapsto_i e[v/x]$.

Here, $[\![(\lambda x{:}\sigma.e)v]\!]$ includes $(\lambda x{:}\sigma'.e')v'$ where $\sigma' \in [\![\sigma]\!]$, $e' \in [\![e]\!]$ and $v' \in [\![v]\!]$. This term steps to $e'[v'/x]$. By Lemma 6, this term is in $[\![e[v/x]]\!]$.

case $(\mathit{fix}\ f{:}\sigma.v_1)v_2 \mapsto_i (v_1[\mathit{fix}\ f{:}\sigma.v_1/f])v_2$

$[\![(\mathit{fix}\ f{:}\sigma.v)v']\!]$ includes terms of the form $(\mathit{fix}\ f{:}\sigma'.v_1')v_2'$ where $\sigma' \in [\![\sigma]\!]$, $v_i' \in [\![v_i]\!]$. This term steps to $(v_1'[\mathit{fix}\ f{:}\sigma'.v_1'/f])v_2'$. By Lemma 6, this term is in $[\![(v_1[\mathit{fix}\ f{:}\sigma.v_1/f])v_2]\!]$.

case $\pi_i\langle v_1, v_2\rangle \mapsto v_i$ Let e in $[\![\pi_i\langle v_1, v_2\rangle]\!]$ be $\pi_i\langle v_1', v_2'\rangle$ for $v_i' \in [\![v_i]\!]$. This term steps to v_i' which is in $[\![v_i]\!]$ by definition.

case

$$\frac{e_1 \mapsto e_1'}{e_1e_2 \mapsto e_1'e_2}$$

Let e in $[\![e_1e_2]\!]$ be $e_1''e_2''$ for $e_i'' \in [\![e_i]\!]$. By induction e_1'' steps to some $e_1''' \in [\![e_1']\!]$. Therefore $e_1''e_2''$ steps to $e_1'''e_2''$ which is in $[\![e_1'e_2]\!]$.

case

$$\frac{e \mapsto e'}{ve \mapsto ve'}$$

Let $e_1 \in [\![ve]\!]$ be $v'e'$ for $v' \in [\![v]\!]$ and $e' \in [\![e]\!]$. By induction e' steps to some $e'' \in [\![e]\!]$. Furthermore, v' is a value. Therefore, $v'e'$ steps to $v'e''$ which is in $[\![ve']\!]$.

case

$$\frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

Here $[\![e[c]]\!]$ includes $e_1[c']e_c$ where $e_1 \in [\![e]\!], \emptyset \vdash c = c' : \kappa, e_c \in \overline{\mathcal{R}[\![c]\!]}$. By induction, e_1 steps to e_2 in $[\![e']\!]$. Therefore $e_1[c']e_c$ steps to $e_2[c']e_c$.

case $\mathit{unpack}\langle\alpha, x\rangle = (\mathit{pack}\ v\ \mathit{as}\ \exists\beta{:}\kappa.\sigma\ \mathit{hiding}\ c)\ \mathit{in}\ e_2 \mapsto e_2[c/\alpha, v/x]$

Here

$$\begin{aligned}[\![e_1]\!] \equiv\ & \mathit{unpack}\langle\alpha, y\rangle = (\mathit{pack}\langle v_c, v'\rangle\ \mathit{as}\ \exists\beta{:}\kappa.R\langle\beta : \kappa\rangle \times \sigma'\ \mathit{hiding}\ c') \\ & \mathit{in}(\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.\lambda x{:}\alpha.e_2')(\pi_1 y)(\pi_2 y)\end{aligned}$$

where $v_c \in \overline{\mathcal{R}[\![c]\!]}, v' \in [\![v]\!], \sigma' \in [\![\sigma]\!], \emptyset \vdash c' = c : \kappa, e_2' \in [\![e_2]\!]$. This term steps to

$$(\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.\lambda x{:}\alpha.e_2')(\pi_1 y)(\pi_2 y)[c'/\alpha][\langle v_c, v'\rangle/y]$$

which steps to

$$e_2'[c'/\alpha][v_c/x_\alpha][v/x]$$

which is in $[\![e_2[c/\alpha][v/x]]\!]$.

case

$$\frac{e \mapsto e'}{\mathit{pack}\ e\ \mathit{as}\ \exists\beta.\sigma\ \ \mathit{hiding}\ c \mapsto \mathit{pack}\ e'\ \mathit{as}\ \exists\beta.\sigma\ \ \mathit{hiding}\ c}$$

This case follows by induction.

case

$$\frac{e \mapsto e'}{\mathit{unpack}\langle\alpha, x\rangle = e\ \mathit{in}\ e_2 \mapsto \mathit{unpack}\langle\alpha, x\rangle = e'\ \mathit{in}\ e_2}$$

This case follows by induction.

□

Lemma 3.4.14 *If* $\vdash e : \mathit{int}$ *and* $e \mapsto_i^* i$ *then for all* $e' \in [\![e]\!]$, $e' \mapsto_R^* i$.

Proof

By induction on the number of steps in $e \mapsto_i^* i$. If e is i and $[\![e]\!]$ is also i. Otherwise, assume $e \mapsto_i e' \mapsto_i^* i$, and let $e_1 \in [\![e]\!]$ be arbitrary. By the previous lemma $e_1 \mapsto e_1' \in [\![e']\!]$, and by induction $e_1' \mapsto^* i$. □

Now we can conclude the dynamic correctness of the translation:

Theorem 3.4.15 (Dynamic Correctness) *If* $\vdash e : \mathit{int}$ *and* $e \mapsto_i^* i$ *then* $|e| \mapsto_R^* i$.

Proof

Special case of the previous lemma. □

3.5 Discussion and chapter summary

In this chapter, I have described the LIR language that enforces the phase distinction between types and terms. Types are only used to describe code, and all information necessary for execution is a part of the term language. The necessary device is a set of terms that represent the type system, and a special singleton type that describes the dependency between the value of these terms and their types. Therefore, the mechanisms of LIR can be applied to type analysis into low-level typed languages.

The ideas of this chapter were used by Hicks, Weirich and Crary (HWC) [HWC01] to add dynamic linking to Typed Assembly Language (TAL) [MWCG99]. In that system, types describe the target language of a type-directed compiler. Because all output of this compiler may be type checked, there is a partial guarantee of the correctness of the compiler. The type system also provides a way for code consumers to verify that the provided programs satisfy critical safety properties. By type checking target code, they do not need to trust the compiler that produced that code, thereby reducing the *trusted computing base*.

In HWC's extension of TAL, the desire was to keep the increase of the trusted computing base to a minimum. In order to add the full capabilities of LIR to typed assembly language, they would have had to add term representations for every element of the large and complicated type language of TAL. The implementation of this addition would have been complicated and its type soundness (though a straightforward extension of the type soundness of LIR) would have had to be proved. Furthermore, any extensions to the type language of TAL would also have to be reflected into type representations—requiring additional trusted implementation.

Instead, they use existing functionality already within the trusted computing base for interpreting binary descriptions of types to form the type representations. The limitation of this strategy is that the creation of representations of types can only occur at compile time. TAL programs cannot dynamically create representations at run time (through some sort of type passing) for arbitrary types, as may be done in LIR. This limitation also prohibits the inclusion of *typecase* or *typerec* in TAL, as run-time type representations cannot be decomposed into smaller parts. However, types may be examined in their entirety. HWC add a checked type casting operation to support the implementation of dynamic types. The addition of this primitive, which behaves the same as the *cast* example from the previous chapter, does not significantly increase the trusted computing base because comparing types for equality is already an operation of the TAL type checker.

In the next chapter, I will discuss an alternative to encoding specialized type representations within the term language. This alternative avoids this unwanted expansion of the trusted computing base and duplication of the type system within the term language. It is possible to extend the expressiveness of the type constructor language so that it may encode a low-level type system such as TAL, using programming language elements such as inductive datatypes and case analysis. By interpreting this encoding at the term level, we may *program* type analysis. What is important about this strategy is that the technical machinery needed for type analysis is independent of the actual type system of the language.

Chapter 4

Type analysis without hard-wired types (I)

4.1 Introduction

In this chapter, I discuss a new approach to adding type analysis to an existing typed programming language. The LX language of Crary and Weirich [CW99a] has a very expressive type constructor language, including elements commonly found in functional programming languages such as products, sums and primitive recursion. Instead of using *typerec* to analyze the *types* LX may only determine whether a sum constructor is a left or right branch. However, with these elements, if the type system of a language (such as LI) may be expressed as inductive datatype, it and analysis over it may be encoded in LX. This language demonstrates that type analysis may be added to a programming language without specializing it directly to the type system of that language. This encoding is important because it separates the mechanism for type analysis from the other features of the language.

4.1.1 Type analysis in typed compilation

An extremely important motivation for the LX language is to support intensional type analysis in the framework of typed-directed compilation. A type-directed compiler operates over a series of typed intermediate languages. Each phase translates the types and terms of the source language into its target in a manner that preserves typing.

However, it is problematic to translate a type-analyzing term from one type language into another. Many translations, such as conversion to continuation-passing style [App92] or closure conversion [MMH96], require a substantial translation of the types. In this case, since the argument to *typecase* has been modified, it is difficult to preserve the meaning of a *typecase* expression. If the type translation is not injective it is impossible to produce the same type analysis, as *typecase* cannot discriminate between two source types that map to the same target. Even if this is not the case, problems still arise. If the transformed types are larger, as is typical, the target analysis must do additional and unnecessary examination to produce the same result. Furthermore, the translation may not be surjective. In this event, exhaustive *typecase*s in the target language do not produce exhaustive *typecase*s in the result, leading to wasteful additional branches that must be inserted by the compiler.

Crary and Weirich proposed the LX language as a solution to this problem. This language allows two distinct notions of type to coexist: the current types and the types used in some earlier stage of compilation. For example, consider the following example of LI *typecase*.

```
Λα:⋆.λx:α.
  typecase α of
    int    =>...(* x has type int *)...
    β × γ  =>...(* x has type β × γ *)...
    β → γ  =>...(* x has type β → γ *)...
```

Now suppose the compiler performs typed closure conversion [MMH96, MWCG99], transforming function types $\tau_1 \to \tau_2$ into $\exists\delta.((\delta \times \tau_1) \to \tau_2) \times \delta$. In LI, `typecase` must add an additional branch.[1]

```
Λα:⋆.λx:α.
  typecase α of
    int                  =>...(* x has type int *)...
    β × γ                =>...(* x has type β × γ *)...
    ∃δ.((δ × β) → γ) × δ =>...(* x has type ∃δ.((δ × β) → γ) × δ *)...
    _                    =>...(* x has some other type *)...
```

Intuitively, we would like α to be a "high-level" type, but upon finding it to be $\beta \to \gamma$ we want to be able to conclude that x has the closure-converted type. The LX language supports the description of several kinds of type. For this example, LX could allow the definition of a special kind *MLType*, representing the types before closure conversion. The translation between *MLType* and the native types of LX may be expressed with a function *interp* : *MLType* $\to \star$, but type analysis may be performed over the members of *MLType*, as below.

[1] This example is not exactly valid in LI, as that language cannot analyze existential types.

$\Lambda\alpha{:}\mathit{MLType}.\, \lambda x{:}(\mathit{interp}\ \alpha).$
```
  typecase α of
    [int]_ML    =>...(* x has type int *)...
    [β × γ]_ML  =>...(* x has type (interp β) × (interp γ) *)...
    [β → γ]_ML  =>...(* x has type ∃δ.((δ × (interp β)) → (interp γ)) × δ *)...
```

LX makes this solution possible by providing a rich programming language of type constructors. In this language, we may define the kind *MLType* using sum, product and inductive kinds, and the operator *interp* using primitive recursion. Section 4.3 demonstrates this definition.

4.1.2 Type analysis as a programming idiom

Although LX was devised to support type analysis, it does not specify the structure of types that may be analyzed. This fact about LX reveals that intensional type analysis is simply a *programming idiom* that is possible in a language with sufficiently rich type constructors.

Crary and Weirich also use this flexibility to extend the capabilities of intensional type analysis by describing how to conduct it in the presence of polymorphic types and other types with binding structure. I will cover this in more detail in Chapter 4.3. In their paper, they also show how to implement "shallow" type analysis, for applications that do not require full type information. They also describe an elegant way to express Haskell-style type classes [PH99] or ML equality types.

Furthermore, in the last chapter I argued that it is important for a type-analyzing language to support a type-erasure semantics. For simplicity, I first present LX with a type-passing semantics. Later in this chapter, I explain the modifications to LX necessary to support a type erasure semantics and the encoding of the type-passing LX into that version. Just as LX can encode the type system with the type constructor language, in the erasure version of LX, the type representations are also definable within the term language.

4.1.3 Informal presentation

I begin the description of the LX language with an example from Crary and Weirich [CW99a] that optimizes memory usage in a polymorphic language. Suppose we wish to store arrays of pairs efficiently. In a naive implementation, because the operations over arrays are polymorphic over their elements, those elements must be the same size. Consequently, each pair in the array must be boxed so that all entries are all word sized. This format requires an additional word for each array entry. It is more efficient to store such arrays using arrays of pairs instead of pairs of arrays.[2]

For functions that manipulate arrays polymorphically, we must use intensional type analysis. Because such a polymorphic array may be actually be a pair of arrays, we must determine the actual type of the array elements before we may access them. To make what we mean concrete, we will first implement this optimization in LI, and then translate it into LX.

To implement this optimization, we define a type operator `optarray` and a corresponding subscript function `optsub` for optimized arrays. The `optarray` operator recursively splits arrays of pairs into pairs of arrays. If the element type is not a pair, it defaults to an ordinary array. (Recursion is not needed at arrow and array types; we assume optimization in those cases is handled by the caller.) The built-in function `sub` has type `forall a.  array a -> int -> a`.

```
type optarray a =
  Typecase a of
    int     => array int
  | b * c   => (optarray b) * (optarray c)
  | b -> c  => array (b -> c)
  | array b => array (array b)

fun optsub[a] (x : optarray a) (n : int) =
   typecase a of
     b * c  => (optsub[b] (#1 x) n, optsub[c] (#2 x) n)
   | _      => sub[a] x n
```

In an LX version of this example, `optarray` and `optsub` no longer operate on types. Instead they operate on *type constructors* that encode the types. For example, we inductively define a kind `MLType` whose members specify the abstract syntax of a type. In this section we use an informal notation borrowed from ML datatypes; we will show how this example is formalized in the next section.

```
kind MLType = Int
            | Prod of MLType * MLType
            | Arrow of MLType * MLType
            | Array of MLType
```

Members of the kind `MLType` have no built-in interpretation as types; they are merely data that may be processed at the level of type constructors. In order to

[2]An ever better representation would be to use arrays of unboxed, flattened tuples. This also can be done straightforwardly using type analysis [HM95], but is a more complicated example.

use them as types, we must define their meaning by a function mapping `MLType` to $\star$:

```
interp (Int)           = int
interp (Prod(c1,c2))   = (interp c1) * (interp c2)
interp (Arrow(c1,c2))  = (interp c1) -> (interp c2)
interp (Array(c))      = array (interp c)
```

Note that the function `interp` is *primitive recursive*. It only calls itself recursively on smaller subcomponents. In order to ensure that computations with type constructors always terminate, arbitrary recursive functions are not permitted in the constructor language of LX. This restriction allows us to use the same method for determining type equality in LX as we used for LI. We will be able to reduce each type to its normal form. Therefore, type checking is decidable in LX.

In LX, we define the new operator `Optarray` of kind `MLType -> MLType` using primitive recursion.

```
Optarray(Int)           = Array(Int)
Optarray(Prod(c1,c2))   = Prod(Optarray(c1), Optarray(c2))
Optarray(Arrow(c1,c2))  = Array(Arrow(c1,c2))
Optarray(Array(c))      = Array(Array(c))
```

The corresponding subscript function, `optsub`, now analyzes members of `MLType` rather than actual types.

```
fun optsub [a : MLType] (x : interp (OptArray a)) (n : int) =
   ccase a of
     Prod(b,c)  => (optsub[b] (#1 x) n, optsub[c] (#2 x) n)
   | _          => sub [interp a] x n
```

Translating this example into LX has certainly made it more verbose, but it also makes it robust under further compilation. Suppose the compiler performs closure conversion, thereby transforming function types $\tau_1 \to \tau_2$ into $\exists\delta.((\delta \times \tau_1) \to \tau_2) \times \delta$. All that is necessary is a change to the appropriate clause of the `interp` function.

```
interp (Arrow (c1, c2))
   = exists d. ((d * interp c1) -> interp c2) * d
```

4.2 A Language for flexible type analysis

In this section, I discuss the formal syntax and semantics of LX. I present the constructor and term levels individually, concentrating discussion on the novel features of each. Like LI and LIR, the syntax of LX (shown in Tables 4.1 and 4.2) is based on Girard's F_ω [Gir71, Gir72]. The difference is that, instead of including built-in constructs for analyzing types, LX includes a rich programming language at the constructor level, and constructor refinement operators at the term level. The full static and operational semantics of LX appear in Section 4.2.3 and in Table 4.3.

4.2.1 Kinds and Constructors

Table 4.1: LX: Syntax for kinds and constructors

κ	$::= \star \mid 1 \mid \kappa_1 \to \kappa_2 \mid \kappa_1 \times \kappa_2 \mid \kappa_1 + \kappa_2$	
	$\mid \chi \mid \mu\chi.\kappa$	
c, τ	$::= * \mid \alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2$	unit, vars and functions
	$\mid \langle c_1, c_2\rangle \mid \pi_1 c \mid \pi_2 c$	products
	$\mid \mathit{inj}_1^{\kappa_1+\kappa_2}\, c \mid \mathit{inj}_2^{\kappa_1+\kappa_2}\, c$	sums
	$\mid \mathit{case}(c, \alpha_1.c_1, \alpha_2.c_2)$	
	$\mid \mathit{fold}_{\mu\chi.\kappa}\, c \mid \mathit{pr}(\chi, \alpha{:}\kappa, \beta{:}\chi \to \kappa'.c)$	primitive recursion
	$\mid \mathit{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2$	types
	$\mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau$	
	$\mid \mathit{unit} \mid \mathit{void} \mid \mu_\kappa(c_1, c_2)$	types

The constructor and kind levels, shown in Table 4.1, contain both base constructors of kind $\star$ (called types) for classifying terms and a variety of programming constructs for computing types. In addition to the type functions of F_ω, LX also includes unit, product, and sum kinds. The introduction and elimination constructors for those kinds are similar to those of the term language, discussed in Chapter 2. We label a few constructors (inj_i, *fold*, *pr*, and μ) with kinds to assist in kind checking; we will omit such kinds when they are clear from context.

Unlike LI and LIR, this language is impredicative and makes no distinction between types and type constructors. Therefore the language requires fewer redundant constructs (i.e. both an $\to$ constructor and an $\to$ type), as both the facilities for statically computing types and the descriptors of the term language occur in the same syntactic category. The *typerec* term of the LI and LIR languages had a restricted domain—it could not analyze polymorphic types. Therefore, those languages used predicativity to restrict what types may be abstracted. LX, on the other hand, does not include any terms that perform type analysis, and so does not need such a restriction.

To support encodings of type structure with abstract syntax trees, LX includes kind variables (χ) and inductive kinds ($\mu\chi.\kappa$). An inductive kind is similar to standard recursive type with the restriction that χ appears only positively within κ. Inductive kinds are formed using the introductory operator $\mathit{fold}_{\mu\chi.\kappa}$, which coerces constructors from kind $\kappa[\mu\chi.\kappa/\chi]$ to kind $\mu\chi.\kappa$. For example, consider the kind of natural numbers Nat, defined as $\mu\chi.(1+\chi)$. The constructor $(\mathit{inj}_1^{1+\mathit{Nat}} *)$ has kind $(1+\chi)[\mathit{Nat}/\chi]$. Therefore $\mathit{fold}_{\mathit{Nat}}(\mathit{inj}_1^{1+\mathit{Nat}} *)$ has kind Nat, and represents the natural number 0.

Inductive kinds are eliminated using the primitive recursion operator pr. Intuitively, $\mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)$ may be thought of as a recursive function with domain $\mu\chi.\kappa$. Within the body of the function c, α is bound to the argument and φ recursively refers to the full pr expression. To ensure that the reduction of constructor expressions always terminates, pr may only define primitive recursive functions. Intuitively, a function is primitive recursive if it can only call itself recursively on a subcomponent of its argument. Following Mendler [Men91], Crary and Weirich [CW99b] enforce this property using abstract kind variables. Since α stands for the unfolded argument, we could consider it to have the kind $\kappa[\mu\chi.\kappa/\chi]$. Instead of substituting for χ in κ, χ is abstract. The recursive variable φ is given kind $\chi \to \kappa'$ (instead of $\chi[\mu\chi.\kappa/\chi] \to \kappa'$) ensuring that φ may be applied only to a subcomponent of α.

The kind κ' in $\mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)$ is permitted to contain (positive) free occurrences of χ. Therefore, the result kind of the above constructor is $\kappa'[\mu\chi.\kappa/\chi]$. This substitution is useful so that some part of the argument may be passed through without φ operating on it. For example, we can define a constructor $\mathit{unfold}_{\mu\chi.\kappa}$ with kind $\mu\chi.\kappa \to \kappa[\mu\chi.\kappa/\chi]$ to be $\mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa.\alpha)$.

Given a constructor n with kind Nat, we can use primitive recursion to construct the type of $(n+1)$-tuples of integers:

$$\begin{array}{lcl} \mathit{ntuple} & \stackrel{\text{def}}{=} & \mathit{pr}(\chi, \alpha{:}1+\chi, \varphi{:}\chi \to \star. \\ & & \quad \mathit{case}\ \alpha\ \mathit{of} \\ & & \quad\quad \mathit{inj}_1\,\beta \Rightarrow \mathit{int} \\ & & \quad\quad \mathit{inj}_2\,\gamma \Rightarrow \varphi(\gamma) \times \mathit{int}) \end{array}$$

Suppose we apply ntuple to $\overline{1}$, that is, the encoding of the natural number 1,

$$\mathit{fold}(\mathit{inj}_2(\mathit{fold}(\mathit{inj}_1 *))).$$

By unrolling the *pr* expression, we may show :

$$
\begin{array}{l}
(pr(\chi, \alpha{:}1 + \chi, \varphi{:}\chi \to \star. \\
\quad case\ \alpha\ of \\
\qquad inj_1\, \beta \Rightarrow int \\
\qquad inj_2\, \gamma \Rightarrow \varphi(\gamma) \times int))\, \overline{1} \\
\\
= case\ (inj_2(fold(inj_1 *)))\ of \\
\qquad inj_1\, \beta \Rightarrow int \\
\qquad inj_2\, \gamma \Rightarrow ntuple(\gamma) \times int \\
\\
= (ntuple(fold(inj_1 *))) \times int \\
\\
= (case\ (inj_1 *)\ of \\
\qquad inj_1\, \beta \Rightarrow int \\
\qquad inj_2\, \gamma \Rightarrow ntuple(\gamma) \times int) \times int \\
\\
= int \times int
\end{array}
$$

The following constructor equivalence rule formalizes the unrolling process for *pr* constructors. The relevant judgment forms of LX are similar to those of LI, and are summarized in Table 4.8:

$$
[ce\text{-}\mu\beta]\quad
\frac{\begin{array}{cc}
\Delta \vdash c' : \kappa[\mu\chi.\kappa/\chi] & \Delta, \chi \vdash \kappa' \\
\Delta, \chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa' \vdash c : \kappa' & \Delta \vdash \mu\chi.\kappa \\
(\chi \text{ only positive in } \kappa' \text{ and } \chi, \alpha, \varphi \notin \Delta) &
\end{array}}
{\begin{array}{l}
\Delta \vdash pr(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)(fold_{\mu\chi.\kappa}\, c') = \\
\qquad c[\mu\chi.\kappa/\chi, c'/\alpha, pr(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)/\varphi] : \kappa'[\mu\chi.\kappa/\chi]
\end{array}}
$$

4.2.2 Terms

The syntax of LX terms appears in Table 4.2. Many LX terms, including the introduction and elimination forms for functions, products, sums, unit, universal and existential types and parameterized recursive types, are from the core language of Section 2.2.2. As in LIR, constructor abstractions are limited by a value restriction, in anticipation of the type erasure interpretation in Section 4.4. The value forms of LX are shown at the bottom of Table 4.2. Also like LIR, recursive functions are expressible using *fix* terms, the bodies of which are syntactically restricted to be functions (possibly polymorphic) by their typing rule (Table 4.7).

Table 4.2: LX: Syntax for terms and values

$e ::= i \mid () \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2$	ints, unit, abstractions
$\mid\ \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e$	products
$\mid\ inj_1^{\tau_1+\tau_2}\, e \mid inj_2^{\tau_1+\tau_2}\, e$	sums
$\mid\ case(e, x_1.e_1, x_2.e_2)$	
$\mid\ \Lambda\alpha{:}\kappa.v \mid e[c] \mid fix\ f{:}\tau.e$	type abstractions, recursion
$\mid\ pack\langle c, e\rangle\ as\ \exists\alpha{:}\kappa.\tau$	existential packages
$\mid\ unpack\langle\alpha, x\rangle = e_1\ in\ e_2$	
$\mid\ roll_{\mu_k(c,c')}\, e \mid unroll\ e$	parameterized recursive types
$\mid\ let[\tau]\,\langle\beta,\gamma\rangle = c\ in\ e$	constructor refinement
$\mid\ let[\tau]\,(fold\ \beta) = c\ in\ e$	
$\mid\ ccase[\tau](c, \alpha_1.e_1, \alpha_2.e_2)$	
$v ::= i \mid () \mid \lambda x{:}c.e \mid \langle v_1, v_2\rangle$	
$\mid\ inj_1^{\tau_1+\tau_2}\, v \mid inj_2^{\tau_1+\tau_2}\, v$	
$\mid\ \Lambda\alpha{:}\kappa.v \mid fix\ f{:}\tau.v \mid roll_{\mu_k(c,c')}\, v$	
$\mid\ pack\ v\ as\ \exists\alpha.c_1\ \ hiding\ c_2$	

Table 4.3: LX: Operational semantics of refinement terms

$$[\textit{ev-ccase1}] \quad \frac{c \text{ normalizes to } \mathit{inj}_1\, c'}{\mathit{ccase}(c, \alpha_1.e_1, \alpha_2.e_2) \mapsto e_1[c'/\alpha_1]}$$

$$[\textit{ev-ccase2}] \quad \frac{c \text{ normalizes to } \mathit{inj}_2\, c'}{\mathit{ccase}(c, \alpha_1.e_1, \alpha_2.e_2) \mapsto e_2[c'/\alpha_2]}$$

$$[\textit{ev-let-prod}] \quad \frac{c \text{ normalizes to } \langle c_1, c_2\rangle}{\mathit{let}\langle\beta, \gamma\rangle = c \;\mathit{in}\; e \mapsto e[c_1, c_2/\beta, \gamma]}$$

$$[\textit{ev-let-fold}] \quad \frac{c \text{ normalizes to } \mathit{fold}_{\mu\chi.\kappa}\, c'}{\mathit{let}(\mathit{fold}_{\mu\chi.\kappa}\, \beta) = c \;\mathit{in}\; e \mapsto e[c'/\beta]}$$

Refinement The novel features of the LX term language are the three refinement operations. To perform constructor analysis at run time, we require a mechanism for branching on sum kinds at the term level. The *ccase* construct supports this branching. For example, if c normalizes to $\mathit{inj}_1(c')$, then the term $\mathit{ccase}(c, \alpha_1.e_1, \alpha_2.e_2)$ evaluates to $e_1[c'/\alpha_1]$.

However, we require more than a term with this evaluation behavior. After branching, we have learned something about the constructor in question, and this information may result in additional knowledge about the types of our data. We wish the type system to be able to exploit that knowledge. Consequently, the typing rule for *ccase*, when the constructor argument is some variable α, substitutes for α to propagate the new information:

$$[\textit{e-ccase}] \quad \frac{\begin{array}{c}\Delta, \beta{:}\kappa_1; \Gamma[\mathit{inj}_1\, \beta/\alpha] \vdash e_1[\mathit{inj}_1\, \beta/\alpha] : \tau[\mathit{inj}_1\, \beta/\alpha] \\ \Delta, \beta{:}\kappa_2; \Gamma[\mathit{inj}_2\, \beta/\alpha] \vdash e_2[\mathit{inj}_2\, \beta/\alpha] : \tau[\mathit{inj}_2\, \beta/\alpha] \\ \Delta, \alpha{:}\kappa_1 + \kappa_2 \vdash c = \alpha : \kappa_1 + \kappa_2\end{array}}{\Delta, \alpha{:}\kappa_1 + \kappa_2; \Gamma \vdash \mathit{ccase}[\tau](c, \beta.e_1, \beta.e_2) : \tau} \quad (\beta \notin \Delta)$$

After substitution, types that once depended upon α are now equivalent to new types, and these types may be different for each branch. For example, if x has type $\mathit{case}(\alpha, \beta.\, \mathit{int}, \beta.\, \mathit{bool})$, its type can be reduced in either branch, allowing it to be used as an integer in one branch and as a boolean in the other.

In order for LX to enjoy the subject reduction property, we also require two *trivialization* rules [CWM02] for *ccase*, for use when the argument to *ccase* is a sum introduction:

$$[\textit{e-triv1}]\quad \frac{\Delta \vdash c = \mathit{inj}_1\, c' : \kappa_1 + \kappa_2 \qquad \Delta; \Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta; \Gamma \vdash \mathit{ccase}[\tau](c, \alpha.e_1, \alpha.e_2) : \tau}$$

$$[\textit{e-triv2}]\quad \frac{\Delta \vdash c = \mathit{inj}_2\, c' : \kappa_1 + \kappa_2 \qquad \Delta; \Gamma \vdash e_2[c'/\alpha] : \tau}{\Delta; \Gamma \vdash \mathit{ccase}[\tau](c, \alpha.e_1, \alpha.e_2) : \tau}$$

Path refinement In the case when the argument to *ccase* is not a variable, we may still like to do refinement. For example, suppose α has kind $(1+1) \times \star$ and x has type $\mathit{case}(\pi_1\alpha, \beta.\, \mathit{int}, \beta.\, \mathit{bool})$. When branching on $\pi_1\alpha$, we should again be able to consider x an integer or boolean, but the ordinary *ccase* rule above no longer applies since $\pi_1\alpha$ is not a variable. To support refinement in this situation, LX includes the product refinement operation, $\mathit{let}[\tau]\,\langle\beta,\gamma\rangle = \alpha\; \mathit{in}\, e$. Like *ccase*, the product refinement operation substitutes everywhere for α:

$$[\textit{e-prod}]\quad \frac{\begin{array}{c}\Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2; \Gamma[\langle\beta,\gamma\rangle/\alpha] \vdash e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \\ \Delta, \alpha{:}\kappa_1 \times \kappa_2 \vdash c = \alpha : \kappa_1 \times \kappa_2\end{array}}{\Delta, \alpha{:}\kappa_1 \times \kappa_2; \Gamma \vdash \mathit{let}[\tau]\langle\beta,\gamma\rangle = c\; \mathit{in}\, e : \tau}\quad (\beta, \gamma \notin \Delta)$$

A similar refinement operation exists for inductive types. Each operation also has trivialization and nonrefining rules similar to those of *ccase*.

We may use these refinement operations to turn paths into variables. For example, suppose α has kind $\mathit{Nat} \times \mathit{Nat}$ and we wish to branch on $\mathit{unfold}(\pi_1\alpha)$. We do so using product and inductive kind refinement:

$$\begin{array}{l} \mathit{let}\,\langle\beta_1,\beta_2\rangle = \alpha\; \mathit{in} \\ \mathit{let}\,(\mathit{fold}\,\gamma) = \beta_1\; \mathit{in} \\ \qquad \mathit{ccase}(\gamma, \delta.e_1, \delta.e_2) \end{array}$$

Nonpath refinement Since there is no refinement operation for functions, sometimes a constructor cannot be reduced to a path. Nevertheless, it is still possible to gain some of the benefits of refinement, using a device due to Harper and Morrisett [HM95]. Suppose φ has kind $\mathit{Nat} \to (1+1)$, x has type $\mathit{case}(\varphi(\overline{1}), \beta.\, \mathit{int}, \beta.\, \mathit{bool})$, and we wish to branch on $\varphi(\overline{1})$ to learn the type of x. First we use a constructor abstraction to assign a variable α to $\varphi(\overline{1})$, thereby enabling *ccase*, and then we use an ordinary abstraction to rebind x with type $\mathit{case}(\alpha, \beta.\, \mathit{int}, \beta.\, \mathit{bool})$:

$$(\Lambda\alpha{:}1+1.\,\lambda x{:}\,case(\alpha,\beta.\,int,\beta.\,bool).$$
$$ccase[\tau](\alpha,\beta.e_1,\beta.e_2))\;[\varphi(\overline{1})]\;x$$

Within e_1, x will be an integer, and similarly within e_2, x will be a Boolean. This device has all the expressive power of refinement, but is less efficient because of the need for extra beta-expansions. However, this is the best that can be done with unknown functions.

4.2.3 Static semantics

Table 4.4: LX: Static semantics for kinds

$\boxed{\Delta \vdash \kappa}$

[*k-type*] $$\frac{}{\Delta \vdash \star}$$

[*k-triv*] $$\frac{}{\Delta \vdash 1}$$

[*k-var*] $$\frac{}{\Delta, \chi \vdash \chi}$$

[*k-mu*] $$\frac{\Delta, \chi \vdash \kappa}{\Delta \vdash \mu\chi.\kappa} \begin{pmatrix}\chi \text{ only positive in } \kappa \\ \chi \notin \Delta\end{pmatrix}$$

[*k-fn*] $$\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 \to \kappa_2}$$

[*k-sum*] $$\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 + \kappa_2}$$

[*k-prod*] $$\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 \times \kappa_2}$$

Table 4.5: LX: Static semantics for constructor formation

$\boxed{\Delta \vdash c : \kappa}$

[*c-triv*]
$$\frac{}{\Delta \vdash * : 1}$$

[*c-var*]
$$\frac{}{\Delta \vdash \alpha : \Delta(\alpha)}$$

[*c-fn*]
$$\frac{\Delta, \alpha{:}\kappa' \vdash c : \kappa \qquad \Delta \vdash \kappa'}{\Delta \vdash \lambda\alpha{:}\kappa'.c : \kappa' \to \kappa}\ (\alpha \notin \Delta)$$

[*c-app*]
$$\frac{\Delta \vdash c_1 : \kappa' \to \kappa \qquad \Delta \vdash c_2 : \kappa'}{\Delta \vdash c_1 c_2 : \kappa}$$

[*c-prod*]
$$\frac{\Delta \vdash c_1 : \kappa_1 \qquad \Delta \vdash c_2 : \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2}$$

[c-π_1]
$$\frac{\Delta \vdash c : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_1 c : \kappa_1}$$

[c-π_2]
$$\frac{\Delta \vdash c : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_2 c : \kappa_2}$$

[*c-inj1*]
$$\frac{\Delta \vdash c : \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \mathit{inj}_1^{\kappa_1+\kappa_2}\, c : \kappa_1 + \kappa_2}$$

[*c-inj2*]
$$\frac{\Delta \vdash c : \kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \mathit{inj}_2^{\kappa_1+\kappa_2}\, c : \kappa_1 + \kappa_2}$$

[*c-case*]
$$\frac{\begin{array}{l}\Delta \vdash c : \kappa_1 + \kappa_2 \\ \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa \\ \Delta, \alpha{:}\kappa_2 \vdash c_2 : \kappa\end{array}}{\Delta \vdash \mathit{case}(c, \alpha.c_1, \alpha.c_2) : \kappa}\ (\alpha \notin \Delta)$$

[*c-fold*]
$$\frac{\Delta \vdash c : \kappa[\mu\chi.\kappa/\chi]}{\Delta \vdash \mathit{fold}_{\mu\chi.\kappa}\, c : \mu\chi.\kappa}$$

Table 4.5 (Continued)

$[c\text{-}pr]$	$$\frac{\Delta, \chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa' \vdash c : \kappa' \qquad \Delta \vdash \mu\chi.\kappa \qquad \Delta \vdash \mu\chi.\kappa'}{\Delta \vdash pr(\chi, \alpha{:}\kappa, \varphi{:}\chi{\to}\kappa'.c) : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]} \;(\ \chi, \alpha, \varphi \notin \Delta\ )$$
$[c\text{-}int\text{-}type]$	$$\frac{}{\Delta \vdash int : \star}$$
$[c\text{-}fn\text{-}type]$	$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \to \tau_2 : \star}$$
$[c\text{-}prod\text{-}type]$	$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \times \tau_2 : \star}$$
$[c\text{-}sum\text{-}type]$	$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 + \tau_2 : \star}$$
$[c\text{-}all\text{-}type]$	$$\frac{\Delta, \alpha{:}\kappa \vdash \tau : \star \qquad \Delta \vdash \kappa}{\Delta \vdash \forall\alpha{:}\kappa.\tau : \star} \;(\alpha \notin \Delta)$$
$[c\text{-}ex\text{-}type]$	$$\frac{\Delta, \alpha{:}\kappa \vdash \tau : \star \qquad \Delta \vdash \kappa}{\Delta \vdash \exists\alpha{:}\kappa.\tau : \star} \;(\alpha \notin \Delta)$$
$[c\text{-}void\text{-}type]$	$$\frac{}{\Delta \vdash void : \star}$$
$[c\text{-}unit\text{-}type]$	$$\frac{}{\Delta \vdash unit : \star}$$
$[c\text{-}rec\text{-}type]$	$$\frac{\Delta \vdash c : (\kappa \to \star) \to \kappa \to \star \qquad \Delta \vdash \kappa \qquad \Delta \vdash c' : \kappa}{\Delta \vdash \mu_\kappa(c, c') : \star}$$

Table 4.6: LX: Static semantics for constructor equivalence

$\boxed{\Delta \vdash c = c' : \kappa}$

[*ce-μβ*]

$$\frac{\begin{array}{cc} \Delta \vdash c' : \kappa[\mu\chi.\kappa/\chi] & \Delta \vdash \mu\chi.\kappa' \\ \Delta, \chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa' \vdash c : \kappa' & \Delta \vdash \mu\chi.\kappa \\ & (\chi, \alpha, \varphi \notin \Delta) \end{array}}{\begin{array}{l} \Delta \vdash pr(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)(fold_{\mu\chi.\kappa}\, c') = \\ \quad c[\mu\chi.\kappa/\chi, c'/\alpha, pr(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)/\varphi] : \kappa'[\mu\chi.\kappa/\chi] \end{array}}$$

[*ce-π₁β*]

$$\frac{\Delta \vdash c_1 : \kappa \qquad \Delta \vdash c_2 : \kappa'}{\Delta \vdash \pi_1\langle c_1, c_2\rangle = c_1 : \kappa}$$

[*ce-π₂β*]

$$\frac{\Delta \vdash c_1 : \kappa' \qquad \Delta \vdash c_2 : \kappa}{\Delta \vdash \pi_2\langle c_1, c_2\rangle = c_2 : \kappa}$$

[*ce-π_η*]

$$\frac{\Delta \vdash c : \kappa_1 \times \kappa_2}{\Delta \vdash \langle \pi_1 c, \pi_2 c\rangle = c : \kappa_1 \times \kappa_2}$$

[*ce-fnβ*]

$$\frac{\begin{array}{cc} \Delta \vdash \kappa' & \\ \Delta, \alpha{:}\kappa' \vdash c : \kappa' & \Delta \vdash c' : \kappa \end{array}}{\Delta \vdash (\lambda\alpha{:}\kappa'.c)c' = c[c'/\alpha] : \kappa} \ (\alpha \notin \Delta)$$

[*ce-fnη*]

$$\frac{\Delta \vdash c : \kappa' \to \kappa}{\Delta \vdash (\lambda\alpha{:}\kappa'.c\alpha) = c : \kappa' \to \kappa} \ (\alpha \notin c)$$

[*ce-inj₁*]

$$\frac{\begin{array}{cc} \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa & \Delta, \alpha{:}\kappa_2 \vdash c_2 : \kappa \\ \Delta \vdash c : \kappa_1 & \Delta \vdash \kappa_2 \end{array}}{\Delta \vdash case(inj_1^{\kappa_1+\kappa_2}\, c, \alpha.c_1, \alpha.c_2) = c_1[c/\alpha] : \kappa}$$

[*ce-inj₂*]

$$\frac{\begin{array}{cc} \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa & \Delta, \alpha{:}\kappa_2 \vdash c_2 : \kappa \\ \Delta \vdash c : \kappa_2 & \Delta \vdash \kappa_1 \end{array}}{\Delta \vdash case(inj_2^{\kappa_1+\kappa_2}\, c, \alpha.c_1, \alpha.c_2) = c_2[c/\alpha] : \kappa}$$

[*ce-case η*]

$$\frac{\Delta \vdash c : \kappa_1 + \kappa_2}{\begin{array}{l} \Delta \vdash case(c, \alpha_1.\, inj_1^{\kappa_1+\kappa_2}\, \alpha_1, \alpha_2.\, inj_2^{\kappa_1+\kappa_2}\, \alpha_2) = \\ \quad c : \kappa_1 + \kappa_2 \end{array}}$$

Table 4.6 (Continued)

[*ce-refl*]	$\dfrac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa}$
[*ce-sym*]	$\dfrac{\Delta \vdash c' = c : \kappa}{\Delta \vdash c = c' : \kappa}$
[*ce-trans*]	$\dfrac{\Delta \vdash c_1 = c_2 : \kappa \qquad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}$
[*ce-cong-fn*]	$\dfrac{\Delta, \alpha{:}\kappa' \vdash c = c' : \kappa \qquad \Delta \vdash \kappa'}{\Delta \vdash \lambda\alpha{:}\kappa'.c = \lambda\alpha{:}\kappa'.c' : \kappa' \to \kappa}\ (\alpha \notin \Delta)$
[*ce-cong-app*]	$\dfrac{\Delta \vdash c_1 = c_1' : \kappa' \to \kappa \qquad \Delta \vdash c_2 = c_2' : \kappa'}{\Delta \vdash c_1 c_2 = c_1' c_2' : \kappa}$
[*ce-cong-prod*]	$\dfrac{\Delta \vdash c_1 = c_1' : \kappa_1 \qquad \Delta \vdash c_2 = c_2' : \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle = \langle c_1', c_2' \rangle : \kappa_1 \times c_2}$
[*ce-cong-prj1*]	$\dfrac{\Delta \vdash c = c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_1 c = \pi_1 c' : \kappa_1}$
[*ce-cong-prj2*]	$\dfrac{\Delta \vdash c = c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_2 c = \pi_2 c' : \kappa_2}$
[*ce-cong-inj1*]	$\dfrac{\Delta \vdash c = c' : \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \mathit{inj}_1^{\kappa_1+\kappa_2}\, c = \mathit{inj}_1^{\kappa_1+\kappa_2}\, c' : \kappa_1 + \kappa_2}$
[*ce-cong-inj2*]	$\dfrac{\Delta \vdash c = c' : \kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \mathit{inj}_2^{\kappa_1+\kappa_2}\, c = \mathit{inj}_2^{\kappa_1+\kappa_2}\, c' : \kappa_1 + \kappa_2}$
[*ce-cong-case*]	$\dfrac{\begin{array}{c}\Delta \vdash c = c' : \kappa_1 + \kappa_2 \\ \Delta, \alpha{:}\kappa_1 \vdash c_1 = c_1' : \kappa \\ \Delta, \alpha{:}\kappa_2 \vdash c_2 = c_2' : \kappa\end{array}}{\begin{array}{l}\Delta \vdash \mathit{case}(c, \alpha.c_1, \alpha.c_2) = \\ \qquad \mathit{case}(c', \alpha.c_1', \alpha.c_2') : \kappa\end{array}}\ (\alpha \notin \Delta)$

Table 4.6 (Continued)

[*ce-cong-fold*]	$$\frac{\Delta \vdash c = c' : \kappa[\mu\chi.\kappa/\chi]}{\Delta \vdash \mathit{fold}_{\mu\chi.\kappa}\, c = \mathit{fold}_{\mu\chi.\kappa}\, c' : \mu\chi.\kappa}$$
[*ce-cong-pr*]	$$\frac{\begin{array}{c}\Delta, \chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa' \vdash c_1 = c_2 : \kappa' \\ \Delta \vdash \mu\chi.\kappa \qquad \Delta, \chi \vdash \kappa'\end{array}}{\begin{array}{l}\Delta \vdash \mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c_1) = \mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c_2) \\ \quad : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\end{array}}$$ $(\chi$ only positive in κ' and $\chi, \alpha, \varphi \notin \Delta)$
[*ce-cong-fn-type*]	$$\frac{\Delta \vdash \tau_1 = \tau_1' : \star \qquad \Delta \vdash \tau_2 = \tau_2' : \star}{\Delta \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' : \star}$$
[*ce-cong-prod-type*]	$$\frac{\Delta \vdash \tau_1 = \tau_1' : \star \qquad \Delta \vdash \tau_2 = \tau_2' : \star}{\Delta \vdash \tau_1 \times \tau_2 = \tau_1' \times \tau_2' : \star}$$
[*ce-cong-sum-type*]	$$\frac{\Delta \vdash \tau_1 = \tau_1' : \star \qquad \Delta \vdash \tau_2 = \tau_2' : \star}{\Delta \vdash \tau_1 + \tau_2 = \tau_1' + \tau_2' : \star}$$
[*ce-cong-all-type*]	$$\frac{\Delta, \alpha{:}\kappa \vdash \tau = \tau' : \star \qquad \Delta \vdash \kappa}{\Delta \vdash \forall\alpha{:}\kappa.\tau = \forall\alpha{:}\kappa.\tau' : \star} \; (\alpha \notin \Delta)$$
[*ce-cong-ex-type*]	$$\frac{\Delta, \alpha{:}\kappa \vdash \tau = \tau' : \star \qquad \Delta \vdash \kappa}{\Delta \vdash \exists\alpha{:}\kappa.\tau = \exists\alpha{:}\kappa.\tau' : \star} \; (\alpha \notin \Delta)$$
[*ce-cong-rec-type*]	$$\frac{\begin{array}{c}\Delta \vdash \kappa \qquad \Delta \vdash c_2 = c_2' : \kappa \\ \Delta \vdash c_1 = c_1' : (\kappa \to \star) \to \kappa \to \star\end{array}}{\Delta \vdash \mu_\kappa(c_1, c_2) = \mu_\kappa(c_1', c_2') : \star}$$

Table 4.7: LX: Static semantics for expressions

$\boxed{\Delta; \Gamma \vdash e : \tau}$

Table 4.7 (Continued)

[*e-ccase*]	$$\frac{\begin{array}{c}\Delta, \beta{:}\kappa_1; \Gamma[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] \vdash e_1[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] : \tau[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] \\ \Delta, \beta{:}\kappa_2; \Gamma[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] \vdash e_2[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] : \tau[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] \\ \Delta, \alpha{:}\kappa_1+\kappa_2 \vdash c = \alpha : \kappa_1+\kappa_2 \\ (\beta \notin \Delta, \tau, \Gamma)\end{array}}{\Delta, \alpha{:}\kappa_1+\kappa_2; \Gamma \vdash ccase[\tau](c, \beta.e_1, \beta.e_2) : \tau}$$
[*e-prod*]	$$\frac{\begin{array}{c}\Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2; \Gamma[\langle\beta,\gamma\rangle/\alpha] \vdash e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \\ \Delta, \alpha{:}\kappa_1\times\kappa_2 \vdash c = \alpha : \kappa_1\times\kappa_2 \\ (\beta, \gamma \notin \Delta, \tau, \Gamma))\end{array}}{\Delta, \alpha{:}\kappa_1\times\kappa_2; \Gamma \vdash let[\tau]\langle\beta,\gamma\rangle = c\; in\; e : \tau}$$
[*e-fold*]	$$\frac{\begin{array}{c}\Delta, \beta{:}\kappa[\mu\chi.\kappa/\chi]; \Gamma[fold_{\mu\chi.\kappa}\,\beta/\alpha] \vdash e[fold_{\mu\chi.\kappa}\,\beta/\alpha] : \tau[fold_{\mu\chi.\kappa}\,\beta/\alpha] \\ \Delta, \alpha{:}\mu\chi.\kappa \vdash c = \alpha : \mu\chi.\kappa \\ (\beta \notin \Delta, \tau, \Gamma)\end{array}}{\Delta, \alpha{:}\mu\chi.\kappa; \Gamma \vdash let[\tau](fold_{\mu\chi.\kappa}\,\beta) = c\; in\; e : \tau}$$
[*e-triv1*]	$$\frac{\Delta \vdash c = inj_1^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \qquad \Delta; \Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta; \Gamma \vdash ccase[\tau](c, \alpha.e_1, \alpha.e_2) : \tau}$$
[*e-triv2*]	$$\frac{\Delta \vdash c = inj_2^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \qquad \Delta; \Gamma \vdash e_2[c'/\alpha] : \tau}{\Delta; \Gamma \vdash ccase[\tau](c, \alpha.e_1, \alpha.e_2) : \tau}$$
[*e-triv3*]	$$\frac{\Delta \vdash c = \langle c_1, c_2\rangle : \kappa_1\times\kappa_2 \quad \Delta; \Gamma \vdash e[c_1, c_2/\beta, \gamma] : \tau}{\Delta; \Gamma \vdash let[\tau]\,\langle\beta,\gamma\rangle = c\; in\; e : \tau}$$
[*e-triv4*]	$$\frac{\Delta \vdash c = fold_{\mu\chi.\kappa}(c') \quad \Delta; \Gamma \vdash e[c'/\beta] : \tau}{\Delta; \Gamma \vdash let[\tau]\,(fold_{\mu\chi.\kappa}\,\beta) = c\; in\; e : \tau}$$

4.2.4 Properties of LX

The judgments of the static semantics of LX appear in Table 4.8. Because of the presence of kind variables and their positivity restriction, not all syntactic kinds are well formed. Therefore, LX formalizes kind formation and augments Δ with the currently bound kind variables.

Table 4.8: LX: Judgment forms

Judgment	Meaning
$\Delta \vdash \kappa$	κ is a well-formed kind
$\Delta \vdash c : \kappa$	c is a valid constructor of kind κ
$\Delta \vdash c_1 = c_2 : \kappa$	c_1 and c_2 are equal constructors
$\Delta; \Gamma \vdash e : \tau$	e is a term of type τ

Contexts

$$\begin{array}{lcl} \Delta & ::= & \epsilon \mid \Delta, \chi \mid \Delta, \alpha{:}\kappa \\ \Gamma & ::= & \epsilon \mid \Gamma, x{:}\tau \end{array}$$

Like LI and LIR, LX satisfies the important properties of decidable type checking and type safety. For type checking, the challenging part is deciding equality of type constructors. This equality is defined using a normalize and compare method employing a reduction relation extracted from the equality rules in the same manner as in LI.

Lemma 4.2.1 *Reduction of well-formed constructors is strongly normalizing, confluent, preserves kinds, and is respected by equality.*

Strong normalization is proven using Mendler's variation on Girard's method [Men87, Men91]. Given Lemma 4.2.1 it is easy to show the normalize and compare algorithm to be terminating, sound and complete, and decidability of type checking follows in a straightforward manner.

Theorem 4.2.2 (Decidability) *It is decidable whether or not* $\Delta; \Gamma \vdash e : \tau$ *is derivable in LX.*

We say that a term is *stuck* if it is not a value and if no rule of the operational semantics applies to it. Type safety requires that no well-typed term can become stuck:

Theorem 4.2.3 (Type Safety) *If* $\emptyset \vdash e : \tau$ *and* $e \mapsto^* e'$ *then* e' *is not stuck.*

The proof of this theorem is standard, relying on the usual lemmas: Progress, Subject Reduction and Substitution.

4.3 Programming type analysis

In this section, I discuss how to implement type analysis in general with LX. I begin with a specific example: Crary and Weirich's formalization of the `Optarray` example from Section 4.1.3. Crary and Weirich [CW99a] describe number of novel styles of type analysis. For example, LX may provide a version of type classes where the domain of type analyzing terms may be restricted to include only those types for which the operation is defined. The LX language may also encode shallow representations of types, for applications of type analysis where the complete type information is not necessary at run time. Finally, LX provides the first mechanism for type-level analysis of types with binding structure (such as universal, existential or recursive types). Because it will be relevant to Chapter 6, I will discuss this last extension in detail at the end of this section.

The basic idea of the type analysis programming idiom is to use elements of the constructor language to represent types and to define an interpretation function to extract the represented type. Instead of destructing types through *typerec*, type-analyzing functions examine constructors the built-in features of LX.

Recall from Section 4.1.3 the inductive kind `MLType`

```
kind MLType = Int
            | Prod of MLType * MLType
            | Arrow of MLType * MLType
            | Array of MLType
```

and its (primitive-recursive) interpretation function

```
interp (Int)          = int
interp (Prod(c1,c2))  = (interp c1) * (interp c2)
interp (Arrow(c1,c2)) = (interp c1) -> (interp c2)
interp (Array(c))     = array (interp c)
```

If we add an array type constructor to LX for this example, we can formalize these definitions in LX by encoding the datatype definition of *MLType* into a recursive sum of products. Below, if $\kappa_1 = \mu\chi.\kappa'$ let $\kappa_1[\kappa_2]$ abbreviate $\kappa'[\kappa_2/\chi]$.

$$
\begin{array}{lcl}
\mathit{MLType} & \stackrel{\text{def}}{=} & \mu\chi.(1 + ((\chi \times \chi) + ((\chi \times \chi) + \chi))) \\
\mathit{interp} & \stackrel{\text{def}}{=} & \mathit{pr}(\chi, \alpha{:}\mathit{MLType}[\chi], \varphi{:}\chi \to \star. \\
 & & \quad \mathit{case}\ \alpha\ \mathit{of} \\
 & & \quad\quad \mathit{inj}_1\, \beta \Rightarrow \mathit{int} \\
 & & \quad\quad \mathit{inj}_2\, \beta \Rightarrow \\
 & & \quad\quad\quad (\mathit{case}\ \beta\ \mathit{of} \\
 & & \quad\quad\quad\quad \mathit{inj}_1\, \beta \Rightarrow \varphi(\pi_1\beta) \times \varphi(\pi_2\beta) \\
 & & \quad\quad\quad\quad \mathit{inj}_2\, \beta \Rightarrow \\
 & & \quad\quad\quad\quad\quad (\mathit{case}\ \beta\ \mathit{of} \\
 & & \quad\quad\quad\quad\quad\quad \mathit{inj}_1\, \beta \Rightarrow \varphi(\pi_1\beta) \to \varphi(\pi_2\beta) \\
 & & \quad\quad\quad\quad\quad\quad \mathit{inj}_2\, \beta \Rightarrow \mathit{array}(\varphi(\beta)))))
\end{array}
$$

Now recall the function `optsub` from Section 4.1.3. To formalize `optsub` in LX, we use *ccase* and inductive kind refinement:

$$
\begin{array}{l}
\mathit{fix}\ \mathit{optsub} : (\forall\alpha{:}\mathit{MLType}.\ \mathit{interp}(\mathit{OptArray}(\alpha)) \to \mathit{int} \to \mathit{interp}(\alpha)). \\
\quad \Lambda\alpha{:}\mathit{MLType}.\ \lambda x{:}\ \mathit{interp}\ (\mathit{OptArray}(\alpha)).\ \lambda n{:}\ \mathit{int}\,. \\
\quad\quad \mathit{let}\ (\mathit{fold}\ \alpha') = \alpha\ \mathit{in} \\
\quad\quad \mathit{ccase}\ \alpha'\ \mathit{of} \\
\quad\quad\quad \mathit{inj}_1\, \beta \Rightarrow \mathit{sub}[\mathit{interp}(\alpha)]\ x\ n \\
\quad\quad\quad \mathit{inj}_2\, \beta \Rightarrow \\
\quad\quad\quad\quad (\mathit{ccase}\ \beta\ \mathit{of} \\
\quad\quad\quad\quad\quad \mathit{inj}_1\, \gamma \Rightarrow \langle \mathit{optsub}[\pi_1\gamma]\ (\pi_1 x)\ n, \mathit{optsub}[\pi_2\gamma]\ (\pi_2 x)\ n\rangle \\
\quad\quad\quad\quad\quad \mathit{inj}_2\, \gamma \Rightarrow \ldots)
\end{array}
$$

We may verify that `optsub` is well typed using the typing rules from the previous section. The interesting branch is the one dealing with products (beginning with "$\mathit{inj}_1\,\gamma \Rightarrow \ldots$"). The *let* operation creates a new variable α' with kind $\mathit{MLType}[\mathit{MLType}]$ and substitutes $\mathit{fold}(\alpha')$ everywhere that α appears. In the product branch, after two uses of *ccase*, γ has kind $\mathit{MLType} \times \mathit{MLType}$ and $\mathit{inj}_2(\mathit{inj}_1(\gamma))$ is substituted for α'.

The required result type is $\mathit{interp}(\alpha)$, which (after substitution) becomes

$$\mathit{interp}(\mathit{fold}(\mathit{inj}_2(\mathit{inj}_1(\gamma))))$$

which by definition is equal to

$$\mathit{interp}(\pi_1\gamma) \times \mathit{interp}(\pi_2\gamma).$$

The type of x is $interp(OptArray(\alpha))$

$$= interp(OptArray(fold(inj_2(inj_1(\gamma)))))$$
$$= interp(OptArray(\pi_1\gamma)) \times interp(OptArray(\pi_2\gamma)).$$

Thus $\pi_1 x$ and $\pi_2 x$ have the appropriate type for the call to *optsub* and the branch type checks.

4.3.1 Types with binding structure

Because types with binding structure (universal, existential and recursive types) cannot generally be included in an inductive description of the type system, LI prohibited the analysis of those type constructors. However, by coding the abstract syntax of the type, Crary and Weirich provide the first type-level analysis of types with binding structure.

For example, we can encode the polymorphic lambda calculus using de Bruijn indices as follows (because the official LX syntax is so verbose, we will use the ML datatype notation):

```
kind Nat   = Zero
           | Succ of Nat
kind FType = Var of Nat
           | Arrow of FType * FType
           | Forall of FType
```

To interpret an `FType` we also need to provide an environment `env` that maps type variables (natural numbers) to types. Thus `interp` has kind `(Nat -> *) -> FType -> *`. To interpret variables, we retrieve them from the environment. For arrow types, we interpret the subcomponents of the arrow with the same environment. In the `Forall` branch, we interpret the body with an appropriately extended environment.

```
interp env (Var (c))        = env(c)
interp env (Arrow (c1,c2)) = (interp env c1) -> (interp env c2)
interp env (Forall (c))    = forall (a:*).
                                interp (fn (b:Nat) =>
                                   case (unfold b) of
                                     Zero   => a
                                   | Succ n => env n) c
```

Type analysis of this language at the term level can be defined in a manner similar to the previous example.

Table 4.9: LX: Representation types

$$
\begin{aligned}
R\langle c : 1\rangle &\stackrel{\text{def}}{=} \mathit{unit} \\
R\langle c : \kappa_1 \to \kappa_2\rangle &\stackrel{\text{def}}{=} \forall\alpha{:}\kappa_1.R\langle\alpha : \kappa_1\rangle \to R\langle c\alpha : \kappa_2\rangle \\
&\quad\ \text{(where } \alpha \text{ is fresh)} \\
R\langle c : \kappa_1 \times \kappa_2\rangle &\stackrel{\text{def}}{=} R\langle\pi_1 c : \kappa_1\rangle \times R\langle\pi_2 c : \kappa_2\rangle \\
R\langle c : \kappa_1 + \kappa_2\rangle &\stackrel{\text{def}}{=} \mathit{case}(c, \alpha.R\langle\alpha : \kappa_1\rangle, \alpha.\,\mathit{void}) + \\
&\quad\ \mathit{case}(c, \alpha.\,\mathit{void}, \alpha.R\langle\alpha : \kappa_2\rangle) \\
R\langle c : \chi\rangle &\stackrel{\text{def}}{=} \varphi_\chi c \\
R\langle c : \mu\chi.\kappa\rangle &\stackrel{\text{def}}{=} \mu_{\mu\chi.\kappa}(\lambda\varphi_\chi{:}\mu\chi.\kappa \to \star. \\
&\quad\ \ \lambda\alpha{:}\mu\chi.\kappa.R\langle \mathit{unfold}\,\alpha : \kappa\rangle, c) \\
&\quad\ \text{(where } \alpha \text{ is fresh)} \\
R\langle c : \star\rangle &\stackrel{\text{def}}{=} \mathit{unit}
\end{aligned}
$$

4.4 Type erasure

The most important contribution of LIR is its reconciliation of type analysis with type-erasure semantics, through the use of primitive terms that express the representations of types at run time. This mechanism allows a semantics where types and type constructors may be erased, as their representations remain to be examined. As I argued in Chapter 3, a type erasure semantics is essential in extending type analysis to low-level languages. In this section, I describe how the methodology of LIR may be used in a type erasable version of LX called LXR and how LX may be translated to this language.

The key part of the translation between LI to LIR was to replace the analysis of any type by an analysis of its representation. So that we could form these type representations when parts of the type were abstract, it was important that the translation ensure that whenever a term abstracts a type variable it also abstracts the representation of that type.

The translation between LX and LXR is very much analogous to this translation. Again, the most important part is to create term representations of LX type constructors, and replace LX's *ccase* operator with a term analysis of the representation of the argument to *ccase*. To support this translation, LXR includes a special form called *vcase*, discussed below. The static and dynamic semantics for *vcase* appear in Tables 4.11 and 4.12. Because the most important part of this translation is representing the type constructors with LXR terms, the rest of this section is devoted to that definition.

Table 4.10: LX: Representation terms

$$
\begin{aligned}
\mathcal{R}|\,*\,| &\stackrel{\text{def}}{=} () \\
\mathcal{R}|\alpha| &\stackrel{\text{def}}{=} x_\alpha \\
\mathcal{R}|\lambda\alpha{:}\kappa.c| &\stackrel{\text{def}}{=} \Lambda\alpha{:}\kappa.\,\lambda x_\alpha{:}R\langle\alpha:\kappa\rangle.\,\mathcal{R}|c| \\
\mathcal{R}|c_1c_2| &\stackrel{\text{def}}{=} \mathcal{R}|c_1|[c_2]\mathcal{R}|c_2| \\
\mathcal{R}|\langle c_1,c_2\rangle| &\stackrel{\text{def}}{=} \langle\mathcal{R}|c_1|,\mathcal{R}|c_2|\rangle \\
\mathcal{R}|\pi_i c| &\stackrel{\text{def}}{=} \pi_i\mathcal{R}|c| \\
\mathcal{R}|\,\mathit{inj}_i^{\kappa_1+\kappa_2}\,c| &\stackrel{\text{def}}{=} \mathit{inj}_i^{R\langle \mathit{inj}_i\,c:\kappa_1+\kappa_2\rangle}\,\mathcal{R}|c| \\
\mathcal{R}|\,\mathit{case}(c,\alpha.c_1,\alpha.c_2)| &\stackrel{\text{def}}{=} (\Lambda\beta{:}\kappa_1+\kappa_2.\,\lambda x{:}R\langle\beta:\kappa_1+\kappa_2\rangle. \\
&\qquad \mathit{case}\;x\;\mathit{of} \\
&\qquad\quad \mathit{inj}_1\,x_\alpha \Rightarrow \mathit{vcase}[R\langle \mathit{case}(\beta,\alpha.c_1,\alpha.c_2):\kappa\rangle] \\
&\qquad\qquad (\beta,\alpha.\mathcal{R}|c_1|,\alpha.\,\mathit{dead}\,x_\alpha) \\
&\qquad\quad \mathit{inj}_2\,x_\alpha \Rightarrow \mathit{vcase}[R\langle \mathit{case}(\beta,\alpha.c_1,\alpha.c_2):\kappa\rangle] \\
&\qquad\qquad (\beta,\alpha.\,\mathit{dead}\,x_\alpha,\alpha.\mathcal{R}|c_2|)) \\
&\qquad\quad [c]\mathcal{R}|c| \\
&\quad \text{(where } \beta \text{ is fresh, } \kappa_1+\kappa_2 \text{ is the kind of } c \\
&\quad\ \text{and } \kappa \text{ is the kind of } \mathit{case}(c,\alpha.c_1,\alpha.c_2)) \\
\mathcal{R}|\,\mathit{fold}_{\mu\chi.\kappa}\,c| &\stackrel{\text{def}}{=} \mathit{roll}_{R\langle \mathit{fold}_{\mu\chi.\kappa}\,c:\mu\chi.\kappa\rangle}\,\mathcal{R}|c| \\
\mathcal{R}|\,\mathit{pr}(\chi,\alpha{:}\kappa,\varphi{:}\chi\to\kappa'.c)| &\stackrel{\text{def}}{=} \mathit{fix}\,x_\varphi. \\
&\qquad \Lambda\beta{:}\mu\chi.\kappa.\,\lambda x{:}R\langle\beta:\mu\chi.\kappa\rangle. \\
&\qquad\quad (\lambda x_\alpha{:}R\langle \mathit{unfold}\,\beta:\kappa[\mu\chi.\kappa/\chi]\rangle. \\
&\qquad\qquad \mathcal{R}|c| \\
&\qquad\qquad\quad [\mu\chi.\kappa/\chi,(\lambda\gamma{:}\mu\chi.\kappa.R\langle\gamma:\mu\chi.\kappa\rangle)/\varphi_\chi, \\
&\qquad\qquad\quad \mathit{unfold}\,\beta/\alpha,\mathit{pr}(\chi,\alpha{:}\kappa,\varphi{:}\chi\to\kappa'.c)/\varphi]) \\
&\qquad (\mathit{unroll}\,x) \\
&\text{(where } \beta \text{ is fresh)} \\
\mathcal{R}|\,\mathit{int}\,|,\mathcal{R}|\tau_1\to\tau_2|,\ldots &\stackrel{\text{def}}{=} ()
\end{aligned}
$$

To represent the basic type constructors of LI, the LIR language contains special terms (R_{int} , $R_{\to}$, $R_{\times}$). Besides those, it was necessary to define representations for the rest of the LI constructor language, including functions, variables and applications. We did so with the notation $\mathcal{R}|c|$ (see Table 3.6) for the representation of the constructor c. For example, we represented constructor functions by term functions and constructor application by term application.

$$\begin{array}{lcl} \mathcal{R}|\alpha| & \stackrel{\text{def}}{=} & x_\alpha \\ \mathcal{R}|\lambda\alpha{:}\kappa.c| & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\kappa.\ \lambda x_\alpha{:}R\langle\alpha:\kappa\rangle.\ \mathcal{R}|c| \\ \mathcal{R}|c_1 c_2| & \stackrel{\text{def}}{=} & \mathcal{R}|c_1|\ [c_2]\ \mathcal{R}|c_2| \end{array}$$

For each representation, the kind of the constructor determines the type of its representation. We formed the type of the representation $\mathcal{R}|c|$ with the definition of $R\langle c : \kappa\rangle$ in Table 4.9. We added the special R-type to LIR, which formed the type of representations of kind $\star$.

$$R\langle c : \star\rangle = R(c)$$

For constructors of function kind, the representations are of type

$$R\langle c : \kappa_1 \to \kappa_2\rangle \stackrel{\text{def}}{=} \forall\alpha{:}\kappa_1.R\langle\alpha : \kappa_1\rangle \to R\langle c\alpha : \kappa_2\rangle \quad \text{(where } \alpha \text{ is fresh)}$$

Because the LX language includes a rich constructor language with product, sum and inductive constructors, we will need to extend the definitions of $\mathcal{R}|c|$ and $R\langle c : \kappa\rangle$ to include these new forms. As the LX language allows the encoding of various type systems through this rich constructor language, LXR allows the *encoding* of representations of those types. Therefore, we will not need the R-type or the basic terms R_{int} , $R_{\to}$ and $R_{\times}$. By extending the definition of $\mathcal{R}|c|$ to the rich forms of LX we may encode term representations of types with the representations of the constructors we used to encode the types. In Section 4.4.2 we demonstrate this idea with an embedding of LIR within LXR.

The complete definitions of $\mathcal{R}|c|$ and $R\langle c : \kappa\rangle$ for LX appear in Tables 4.10 and 4.9. In general, we represent an LX constructor form with an equivalent LX term form. The reason is one of consistency. If two LX type constructors are equivalent (according to the definition of equality, $\Delta \vdash c_1 = c_2 : \kappa$) then the terms that represent them should behave in equivalent ways.

Because types may not be analyzed in LX, they have trivial representations in LXR. All types are represented by the term (). The representation of constructor functions and application remains the same as in LIR. We map unit constructors to unit terms, and product constructors to product terms.

$$\begin{array}{lcl}\mathcal{R}|\,*\,| & \stackrel{\text{def}}{=} & () \\ \mathcal{R}|\langle c_1, c_2\rangle| & \stackrel{\text{def}}{=} & \langle \mathcal{R}|c_1|, \mathcal{R}|c_2|\rangle \\ \mathcal{R}|\pi_i c| & \stackrel{\text{def}}{=} & \pi_i(\mathcal{R}|c|)\end{array}$$

Therefore, the type of the representations of constructors with unit kinds is *unit* and the type of representations of constructors with product kinds is a product type.

$$\begin{array}{lcl}R\langle c : 1\rangle & \stackrel{\text{def}}{=} & \mathit{unit} \\ R\langle c : \kappa_1 \times \kappa_2\rangle & \stackrel{\text{def}}{=} & R\langle \pi_1 c : \kappa_1\rangle \times R\langle \pi_2 c : \kappa_2\rangle\end{array}$$

Inductive Constructors We represent an inductive constructor c using a recursive type parameterized by c. The recursive type binds the variable φ_χ to compute the representation of the inductive type.

$$\begin{array}{lcl}R\langle c : \mu\chi.\kappa\rangle & \stackrel{\text{def}}{=} & \mu_{\mu\chi.\kappa}(\lambda\varphi_\chi{:}\mu\chi.\kappa \to \star.\ \lambda\alpha{:}(\mu\chi.\kappa).\ R\langle \mathit{unfold}\,\alpha : \kappa\rangle, c) \\ R\langle c : \chi\rangle & \stackrel{\text{def}}{=} & \varphi_\chi\, c\end{array}$$

As *fold* introduces an inductive kind in the constructor language, we use *roll* introduce the appropriate recursive type.

$$\mathcal{R}|\,\mathit{fold}_{\mu\chi.\kappa}\, c| \stackrel{\text{def}}{=} \mathit{roll}_{R\langle \mathit{fold}_{\mu\chi.\kappa}\, c:\mu\chi.\kappa\rangle}\, \mathcal{R}|c|$$

To represent primitive recursion, we must use *fix*, which creates iteration at the term level. Suppose β is fresh. The type of the representation of a *pr* constructor should be:

$$\begin{array}{l}R\langle \mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c) : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle \\ = \forall\beta{:}\mu\chi.\kappa.R\langle \beta : \mu\chi.\kappa\rangle \to R\langle \mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)\beta : \kappa'[\mu\chi.\kappa/\chi]\rangle\end{array}$$

$$\begin{array}{l}\mathcal{R}|\,\mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)| \stackrel{\text{def}}{=} \\ \mathit{fix}\, x_\varphi. \\ \quad \Lambda\beta{:}\mu\chi.\kappa.\ \lambda x{:}R\langle \beta : \mu\chi.\kappa\rangle. \\ \quad\quad ((\lambda x_\alpha{:}R\langle \mathit{unfold}\,\beta : \kappa[\mu\chi.\kappa/\chi]\rangle. \\ \quad\quad\quad \mathcal{R}|c|[\mu\chi.\kappa/\chi, (\lambda\gamma.R\langle\gamma : \mu\chi.\kappa\rangle)/\varphi_\chi, (\mathit{unfold}\,\beta)/\alpha, \mathit{pr}(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)/\varphi]) \\ \quad\quad (\mathit{unroll}\, x))\end{array}$$

Table 4.11: LXR: Static semantics for *vcase*

$$[\textit{e-vc1}]\quad \frac{\begin{array}{c}\Delta, \beta{:}\kappa_1; \Gamma[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] \vdash v[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] : void \\ \Delta, \beta{:}\kappa_2; \Gamma[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] \vdash e[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] : \tau[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] \\ \Delta, \alpha{:}\kappa_1+\kappa_2 \vdash c = \alpha : \kappa_1+\kappa_2\end{array}}{\Delta, \alpha{:}\kappa_1+\kappa_2; \Gamma \vdash vcase[\tau](c, \beta.\ dead\ v, \beta.e) : \tau}\quad (\beta \notin \Delta)$$

$$[\textit{e-vc2}]\quad \frac{\begin{array}{c}\Delta, \beta{:}\kappa_1; \Gamma[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] \vdash e[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] : \tau[inj_1^{\kappa_1+\kappa_2}\,\beta/\alpha] \\ \Delta, \beta{:}\kappa_2; \Gamma[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] \vdash v[inj_2^{\kappa_1+\kappa_2}\,\beta/\alpha] : void \\ \Delta, \alpha{:}\kappa_1+\kappa_2 \vdash c = \alpha : \kappa_1+\kappa_2\end{array}}{\Delta, \alpha{:}\kappa_1+\kappa_2; \Gamma \vdash vcase[\tau](c, \beta.e, \beta.\ dead\ v) : \tau}\quad (\beta \notin \Delta)$$

$$[\textit{e-triv5}]\quad \frac{\Delta \vdash c = inj_1^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \qquad \Delta; \Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta; \Gamma \vdash vcase[\tau](c, \alpha.e_1, \alpha.\ dead\ v) : \tau}$$

$$[\textit{e-triv6}]\quad \frac{\Delta \vdash c = inj_2^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \qquad \Delta; \Gamma \vdash e_2[c'/\alpha] : \tau}{\Delta; \Gamma \vdash vcase[\tau](c, \alpha.\ dead\ v, \alpha.e_2) : \tau}$$

$$[\textit{e-triv7}]\quad \frac{\Delta \vdash c = inj_1^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \qquad \Delta; \Gamma \vdash e_2[c'/\alpha] : void}{\Delta; \Gamma \vdash vcase[\tau](c, \alpha.e_1, \alpha.\ dead\ v) : \tau}$$

$$[\textit{e-triv8}]\quad \frac{\Delta \vdash c = inj_2^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \qquad \Delta; \Gamma \vdash e_1[c'/\alpha] : void}{\Delta; \Gamma \vdash vcase[\tau](c, \alpha.\ dead\ v, \alpha.e_2) : \tau}$$

Table 4.12: LXR: Operational semantics for *vcase*

Value Syntax

$$v ::= \ldots \mid (\mathit{fix}\ f{:}\tau.v)[c_1]\ldots[c_n]$$

Operational Rules Remove the rules ev-fix1, ev-fix2, ev-ccase1, ev-ccase2. Add the following rules:

$$[\mathit{ev\text{-}fix}] \quad (\mathit{fix}\ f{:}c.e)[c_1]...[c_n]v \mapsto (e[\mathit{fix}\ f{:}c.e/f])[c_1]...[c_n]v$$

$$[\mathit{ev\text{-}vcase1}] \quad \frac{c \text{ normalizes to } \mathit{inj}_1\, c'}{\mathit{vcase}(c, \alpha_1.e_1, \alpha_2.\ \mathit{dead}\ v) \mapsto e_1[c'/\alpha_1]}$$

$$[\mathit{ev\text{-}vcase2}] \quad \frac{c \text{ normalizes to } \mathit{inj}_2\, c'}{\mathit{vcase}(c, \alpha_1.\ \mathit{dead}\ v, \alpha_2.e_2) \mapsto e_2[c'/\alpha_2]}$$

Sum Constructors The definition of the representation of sum constructors is the most important and the most subtle of all of the constructors of LX. Unlike LI and LIR, instead of analyzing constructors of kind $\star$ with *typecase*, LX analyzes constructors of sum kind with *ccase*. It is this *ccase* that prevents type erasure in LX; the operation of *ccase* depends on its argument type constructor. In order to create an erasable version of LX, we need to replace *ccase* by something that analyzes term representations of the sum constructors.

What should the term representation of a sum constructor be? It makes sense that we should represent sum constructors by sum terms. If the sum constructor is a left injection of some c', then its term representation should be a left injection of the representation of c':

$$\mathcal{R}|\ \mathit{inj}_i^{\kappa_1+\kappa_2}\, c| \stackrel{\text{def}}{=} \mathit{inj}_i^{R\langle \mathit{inj}_i\, c:\kappa_1+\kappa_2\rangle}\, \mathcal{R}|c|$$

What is the type of such a representation? (i.e., what is $R\langle \mathit{inj}_i\, c : \kappa_1 + \kappa_2\rangle$?) The type of an injection term must be a sum type. Furthermore, if c is $\mathit{inj}_1\, c_1$ then left component of this sum type should be $R\langle c_1 : \kappa_1\rangle$. To enforce that the term representation is inj_1 of the representation of c_1, we make the right component of the sum type *void*. Because the void type contains no values, if a term is of type $R\langle c_1 : \kappa\rangle_1 + \mathit{void}$ it must evaluate to an inj_1 term. Likewise, if c is $\mathit{inj}_2\, c_2$ then the right component of the sum type should be $R\langle c_2 : \kappa_2\rangle$ and the left component of the sum type should be *void*. We may express this entire type using a case analysis of c.

$$R\langle c : \kappa_1{+}\kappa_2\rangle \stackrel{\text{def}}{=} \mathit{case}\, c\ (\alpha.R\langle \alpha : \kappa_1\rangle{+}\mathit{void}, \alpha.\,\mathit{void}\,{+}R\langle \alpha : \kappa_2\rangle) \quad \text{(where } \alpha \text{ is fresh)}$$

However, in practice, there is a problem with the above definition. If c is abstract, the type is not a sum type, so we cannot use case analysis of this representation. Even though both branches of the case analysis produce sum types, our equational theory does not let us conclude that the type is equivalent to a sum type. If this type is not a sum, then we cannot use *case* for terms of this type. So when we do not know the type that is represented by such a term, we cannot use case analysis to find out its identity. Therefore, we use a related definition that commutes the sum and the case analysis.

$$R\langle c : \kappa_1 + \kappa_2\rangle \stackrel{\text{def}}{=} \mathit{case}(c, \alpha.R\langle \alpha : \kappa_1\rangle, \alpha.\,\mathit{void}) + \mathit{case}(c, \alpha.\,\mathit{void}, \alpha.R\langle \alpha : \kappa_2\rangle)$$

We already have a facility for analyzing term sums, the standard *case* expression. Say a term of this type is the argument to *case*. In the first branch, the argument is of type

$$\mathit{case}(c, \alpha.R\langle \alpha : \kappa_1\rangle, \alpha.\,\mathit{void}).$$

Because there are no closed values of type *void*, we know that c must be $\mathit{inj}_1\, c'$ for some c'. In order to reflect this knowledge we add a coercion to the term language called a *virtual case* or *vcase*. The typing judgment for *vcase* for the left branch is below:

$$\frac{\begin{array}{l}\Delta, \beta{:}\kappa_1; \Gamma[\mathit{inj}_1\,\beta/\alpha] \vdash e[\mathit{inj}_1\,\beta/\alpha] : \tau[\mathit{inj}_1\,\beta/\alpha] \\ \Delta, \beta{:}\kappa_2; \Gamma[\mathit{inj}_2\,\beta/\alpha] \vdash v[\mathit{inj}_2\,\beta/\alpha] : \mathit{void} \\ \Delta, \alpha{:}\kappa_1 + \kappa_2 \vdash c = \alpha : \kappa_1 + \kappa_2\end{array}}{\Delta, \alpha{:}\kappa_1 + \kappa_2; \Gamma \vdash \mathit{vcase}[\tau](c, \beta.e, \beta.\,\mathit{dead}\ v) : \tau}\ (\beta \notin \Delta)$$

We call this case *virtual* because we know statically which branch will be taken, the left branch. The formation rules for the left branch are just the same as for *ccase*. In the right branch, or *dead* branch, we must show that had the argument constructor c have been a right injection, we would produce a value, v of type *void*. We list the complete rules for *vcase* in Tables 4.11 and 4.12. In addition to the formation rule for *vcase* above (and its analogue), because it does refinement, we require a number of trivialization rules for *vcase*.

Below, we use *vcase* to construct the representation of a constructor that employs case analysis. Because of refinement, the argument to *vcase* must be statically equal to a type variable. Therefore, we must abstract this argument with a fresh variable β. Suppose $\kappa_1 + \kappa_2$ is the kind of c, and κ the kind of $case(c, \alpha.c_1, \alpha.c_2)$:

$$\begin{array}{l}
\mathcal{R}|\, case(c, \alpha.c_1, \alpha.c_2)| \stackrel{\text{def}}{=} \\
\quad (\Lambda\beta{:}\kappa_1 + \kappa_2.\, \lambda x{:}R\langle\beta : \kappa_1 + \kappa_2\rangle. \\
\qquad case\ x\ of \\
\qquad\quad inj_1\, x_\alpha \Rightarrow vcase[R\langle case(\beta, \alpha.c_1, \alpha.c_2) : \kappa\rangle](\beta, \alpha.\mathcal{R}|c_1|, \alpha.\, dead\ x_\alpha) \\
\qquad\quad inj_2\, x_\alpha \Rightarrow vcase[R\langle case(\beta, \alpha.c_1, \alpha.c_2) : \kappa\rangle](\beta, \alpha.\, dead\ x_\alpha, \alpha.\mathcal{R}|c_2|)\ )\ [c]\ \mathcal{R}|c|
\end{array}$$

4.4.1 Type soundness of constructor representation

Next, we prove that a constructor's representation does represent it, by stating that in an appropriate context, the translation of a constructor (from the translation defined in Table 4.10) has the correct type (as defined in Table 4.9). The result is the analogue to Lemma 3.4.1 of the translation between LI and LIR.

We begin, as usual, with some substitution lemmas:

Lemma 4.4.1

$$R\langle c[c'/\alpha] : \kappa\rangle = R\langle c : \kappa\rangle[c'/\alpha]$$

Proof

By structural induction on κ. □

Lemma 4.4.2 *If φ_χ is not free in c*

$$R\langle c[\kappa'/\chi] : \kappa[\kappa'/\chi]\rangle = R\langle c : \kappa\rangle[\kappa', \lambda\alpha{:}\kappa'.R\langle\alpha : \kappa'\rangle/\chi, \varphi_\chi]$$

Proof

by structural induction on κ. □

Using the definition of $R_{\mathsf{con}}(\cdot)$ and $R_{\mathsf{val}}(\cdot)$ in Table 4.13, we may state the connection between a representation and its type.

Theorem 4.4.3 (Representations) *If $\Delta \vdash c : \kappa$ then*

$$R_{\mathsf{con}}(\Delta); R_{\mathsf{val}}(\Delta) \vdash \mathcal{R}|c| : R\langle c : \kappa\rangle.$$

Proof

For notational convenience, define $R(\Delta) \stackrel{\text{def}}{=} R_{\mathsf{con}}(\Delta); R_{\mathsf{val}}(\Delta)$. Proof is by induction on the derivation of $\Delta \vdash c : \kappa$, and case analysis of the last rule of the derivation. Selected cases appear below:

Table 4.13: Translation of LX contexts

$$
\begin{array}{rcl}
R_{\mathsf{con}}(\emptyset) & \stackrel{\text{def}}{=} & \emptyset \\
R_{\mathsf{con}}(\Delta, \chi) & \stackrel{\text{def}}{=} & R_{\mathsf{con}}(\Delta), \chi, \varphi_\chi{:}\chi \to \star \\
R_{\mathsf{con}}(\Delta, \alpha{:}\kappa) & \stackrel{\text{def}}{=} & R_{\mathsf{con}}(\Delta), \alpha{:}\kappa \\
\\
R_{\mathsf{val}}(\emptyset) & \stackrel{\text{def}}{=} & \emptyset \\
R_{\mathsf{val}}(\Delta, \chi) & \stackrel{\text{def}}{=} & R_{\mathsf{val}}(\Delta) \\
R_{\mathsf{val}}(\Delta, \alpha{:}\kappa) & \stackrel{\text{def}}{=} & R_{\mathsf{val}}(\Delta), x_\alpha{:}R\langle \alpha : \kappa\rangle
\end{array}
$$

case (c-inj1)

$$\frac{\Delta \vdash c : \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \mathit{inj}_1^{\kappa_1+\kappa_2}\, c : \kappa_1 + \kappa_2}$$

As $\mathcal{R}|\,\mathit{inj}_1^{\kappa_1+\kappa_2}\, c|$ is defined as $\mathit{inj}_1^{R\langle \mathit{inj}_1\, c:\kappa_1+\kappa_2\rangle}\, \mathcal{R}|c|$ and $R\langle \mathit{inj}_1\, c : \kappa_1 + \kappa_2\rangle$ is $\mathit{case}(\mathit{inj}_1\, c, \alpha.R\langle\alpha : \kappa_1\rangle, \alpha.\, \mathit{void}) + \mathit{case}(\mathit{inj}_2\, c, \alpha.\, \mathit{void}, \alpha.R\langle \alpha : \kappa_2\rangle)$, or $R\langle c : \kappa_1\rangle + \mathit{void}$, it suffices to show

$$R(\Delta) \vdash \mathit{inj}_1^{R\langle c:\kappa_1\rangle+\mathit{void}}\, \mathcal{R}|c| : R\langle c : \kappa_1\rangle + \mathit{void}$$

By induction we know $R(\Delta) \vdash \mathcal{R}|c| : R\langle c : \kappa_1\rangle$ so we may apply (e-inj1) to derive this result. The case for (e-inj2) is analogous.

case (c-case)

$$\frac{\Delta \vdash c : \kappa_1 + \kappa_2 \qquad \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa \qquad \Delta, \alpha{:}\kappa_2 \vdash c_2 : \kappa}{\Delta \vdash \mathit{case}(c, \alpha.c_1, \alpha.c_2) : \kappa} \; (\alpha \notin \Delta)$$

By induction we know three judgments:

$$R(\Delta) \vdash \mathcal{R}|c| : \mathit{case}(c, \alpha : R\langle\alpha : \kappa_1\rangle, \alpha.\, \mathit{void}) + \mathit{case}(c, \alpha.\, \mathit{void}, \alpha.R\langle\alpha : \kappa_2\rangle)$$

$$R_{\mathsf{con}}(\Delta), \alpha : \kappa_1; R_{\mathsf{val}}(\Delta), x_\alpha : R\langle\alpha : \kappa_1\rangle \vdash \mathcal{R}|c_1| : R\langle c_1 : \kappa_1\rangle$$

$$R_{\mathsf{con}}(\Delta), \alpha : \kappa_2; R_{\mathsf{val}}(\Delta), x_\alpha : R\langle\alpha : \kappa_2\rangle \vdash \mathcal{R}|c_2| : R\langle c_1 : \kappa_2\rangle$$

We wish to show that:

$$R(\Delta) \vdash \mathcal{R}|\,\mathit{case}(c, \alpha.c_1, \alpha.c_2)| : R\langle \mathit{case}(c, \alpha.c_1, \alpha.c_2) : \kappa\rangle$$

By expanding the definition and pushing abstracted variables into the context via (e-fn) (e-tfn) and (e-case), it suffices to show that:

$$\begin{array}{l} R_{\mathsf{con}}(\Delta), \beta : \kappa_1 + \kappa_2; \\ R_{\mathsf{val}}(\Delta), x : R\langle \beta : \kappa_1 + \kappa_2 \rangle, x_\alpha : \mathit{case}(\beta, \alpha.R\langle \alpha : \kappa_1 \rangle, \alpha.\, \mathit{void}) \\ \vdash \mathit{vcase}[R\langle \mathit{case}(\beta, \alpha.c_1, \alpha.c_2) : \kappa \rangle](\beta, \alpha.\mathcal{R}|c_1|, \alpha.\, \mathit{dead}\, x_\alpha) \\ : R\langle \mathit{case}(\beta, \alpha.c_1, \alpha.c_2) : \kappa \rangle \end{array}$$

and its analogue.

First define the term context with β replaced by $\mathit{inj}_1\, \alpha$

$$\begin{array}{rl} \Gamma_1 & \stackrel{\text{def}}{=} (R_{\mathsf{val}}(\Delta), x{:}R\langle \beta : \kappa_1 + \kappa_2 \rangle, x_\alpha{:}\ \mathit{case}(\beta, \alpha.R\langle \alpha : \kappa_1 \rangle, \alpha.\, \mathit{void})) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\mathit{inj}_1\, \alpha/\beta] \\ & \stackrel{\text{def}}{=} R_{\mathsf{val}}(\Delta), x{:}R\langle \alpha : \kappa_1 \rangle + \mathit{void}, x_\alpha{:}R\langle \alpha : \kappa_1 \rangle \end{array}$$

and analogously $\Gamma_2 \stackrel{\text{def}}{=} R_{\mathsf{val}}(\Delta), x{:}\ \mathit{void} + R\langle \alpha : \kappa_2 \rangle, x_\alpha{:}\ \mathit{void}$.

It suffices to show that :

- $(R_{\mathsf{con}}(\Delta), \alpha{:}\kappa_1; \Gamma_1 \vdash \mathcal{R}|c_1| : R\langle \mathit{case}(\mathit{inj}_1\, \alpha), \alpha.c_1, \alpha.c_2 : \kappa \rangle)[\mathit{inj}_1\, \alpha/\beta]$
  As β is not free in c_1 and c_2, this follows from induction.
- $R_{\mathsf{con}}(\Delta), \alpha : \kappa_2; \Gamma_2 \vdash x_\alpha : \mathit{void}$, trivial
- $R_{\mathsf{con}}(\Delta), \beta :\ \kappa_1 + \kappa_2 \vdash \beta = \beta : \kappa_1 + \kappa_2$, trivial

case (c-fold)

$$\frac{\Delta \vdash c : \kappa[\mu\chi.\kappa/\chi]}{\Delta \vdash \mathit{fold}_{\mu\chi.\kappa}\, c : \mu\chi.\kappa}$$

We would like to prove

$$R(\Delta) \vdash \mathit{roll}_{R\langle \mathit{fold}_{\mu\chi.\kappa}\, c:\mu\chi.\kappa \rangle}\, \mathcal{R}|c| : R\langle \mathit{fold}_{\mu\chi.\kappa}\, c : \mu\chi.\kappa \rangle$$

where $R\langle \mathit{fold}_{\mu\chi.\kappa}\, c : \mu\chi.\kappa \rangle = \mu_{\mu\chi.\kappa}(\lambda\varphi_\chi{:}\mu\chi.\kappa \to \star.\lambda\alpha{:}\mu\chi.\kappa.R\langle \mathit{unfold}\, \alpha : \kappa \rangle, \mathit{fold}_{\mu\chi.\kappa}\, c)$

It suffices to prove the premises of (e-fold)

- Well formedness of the recursive type:

$$R_{\mathsf{con}}(\Delta) \vdash \mu_{\mu\chi.\kappa}(\lambda\varphi_\chi{:}\mu\chi.\kappa \to \star.\lambda\alpha{:}\mu\chi.\kappa.R\langle \mathit{unfold}\, \alpha : \kappa \rangle, \mathit{fold}_{\mu\chi.\kappa}\, c) : \star$$

- Correctness of the unfolding:

$$\begin{array}{l} R(\Delta) \vdash \mathcal{R}|c| : R\langle \mathit{unfold}\,\alpha : \kappa\rangle \\ \quad [\mathit{fold}_{\mu\chi.\kappa}\, c/\alpha] \\ \quad [\lambda\alpha{:}\mu\chi.\kappa.\mu_{\mu\chi.\kappa}(\lambda\varphi_\chi{:}\mu\chi.\kappa \to \star.\lambda\beta{:}(\mu\chi.\kappa).R\langle \mathit{unfold}\,\beta : \kappa\rangle, \alpha)/\varphi_\chi] \end{array}$$

  As α is not free in c, we can push the substitution for it through the judgment. Also, as χ is not free in c, we can add a simultaneous substitution.

$$\begin{array}{l} R(\Delta) \vdash \mathcal{R}|c| : R\langle \mathit{unfold}(\mathit{fold}_{\mu\chi.\kappa}\, c) : \kappa\rangle \\ \quad [\mu\chi.\kappa, \lambda\alpha{:}(\mu\chi.\kappa).\mu_{\mu\chi.\kappa}(\lambda\varphi_\chi{:}\mu\chi.\kappa \to \star.\lambda\beta{:}(\mu\chi.\kappa).R\langle \mathit{unfold}\,\beta : \kappa\rangle, \alpha)/\chi, \varphi_\chi] \end{array}$$

  or, using the definition of $R\langle\alpha : \mu\chi.\kappa\rangle$, and the equivalence for *unfold* of *fold*, we may rewrite the judgment as

$$R(\Delta) \vdash \mathcal{R}|c| : R\langle c : \kappa\rangle[\mu\chi.\kappa, \lambda\alpha{:}(\mu\chi.\kappa).R\langle\alpha : \mu\chi.\kappa\rangle/\chi, \varphi_\chi]$$

  By the lemma, it suffices to show: $R(\Delta) \vdash \mathcal{R}|c| : R\langle c : \kappa[\mu\chi.\kappa/\chi]\rangle$ which follows from induction.

case (c-pr)

$$\frac{\Delta, \chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa' \vdash c : \kappa' \qquad \Delta \vdash \mu\chi.\kappa \qquad \Delta, \chi \vdash \kappa'}{\Delta \vdash pr(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c) : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]}\ (\chi \text{ only positive in } \kappa', \text{ and } \chi, \alpha, \varphi \notin \Delta)$$

First, for notational convenience define $\rho \stackrel{\text{def}}{=} pr(\chi, \alpha{:}\kappa, \varphi{:}\chi \to \kappa'.c)$.

We would like to prove that

$$\begin{array}{l} R(\Delta) \vdash \mathit{fix}\, x_\varphi : R\langle \rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle. \\ \qquad \Lambda\beta{:}\mu\chi.\kappa.\ \lambda x{:}R\langle\beta : \mu\chi.\kappa\rangle. \\ \qquad\quad (\lambda x_\alpha{:}R\langle \mathit{unfold}\,\beta : \kappa[\mu\chi.\kappa/\chi]\rangle. \\ \qquad\qquad \mathcal{R}|c|[\mu\chi.\kappa/\chi, (\lambda\gamma{:}\mu\chi.\kappa.R\langle\gamma : \mu\chi.\kappa\rangle)/\varphi_\chi, \mathit{unfold}\,\beta/\alpha, \rho/\varphi]) \\ \qquad\quad (\mathit{unroll}\, x) \\ \qquad : R\langle \rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle \end{array}$$

As $R\langle \rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle$ is equivalent to $\forall\beta{:}\mu\chi.\kappa.R\langle\beta : \mu\chi.\kappa\rangle \to R\langle\rho\beta : \kappa'[\mu\chi.\kappa/\chi]\rangle$, it suffices to show the premises of the application:

$$R(\Delta), x_\varphi{:}R\langle \rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle, \beta{:}\mu\chi.\kappa, x{:}R\langle \beta : \mu\chi.\kappa\rangle$$
$$\vdash (\mathit{unroll}\, x)$$
$$: R\langle \mathit{unfold}\, \beta : \kappa[\mu\chi.\kappa/\chi]\rangle$$

$$R(\Delta), x_\varphi{:}R\langle \rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle, \beta{:}\mu\chi.\kappa, x_\alpha{:}R\langle \mathit{unfold}\, \beta : \kappa[\mu\chi.\kappa/\chi]\rangle$$
$$\vdash \mathcal{R}|c|[\mu\chi.\kappa/\chi, (\lambda\gamma{:}\mu\chi.\kappa.R\langle \gamma : \mu\chi.\kappa\rangle)/\varphi_\chi, \mathit{unfold}\, \beta/\alpha, \rho/\varphi])$$
$$: R\langle \rho\beta : \kappa'[\mu\chi.\kappa/\chi]\rangle$$

The first is by inspection. To show the second, first we know by induction that

$$R_{\mathsf{con}}(\Delta), \chi, \varphi_\chi{:}\chi \to \star, \alpha{:}\kappa, \varphi{:}\chi \to \kappa';$$
$$R_{\mathsf{val}}(\Delta), x_\alpha{:}R\langle \alpha : \kappa\rangle, x_\varphi{:}(\forall\gamma{:}\chi.\varphi_\chi\gamma \to R\langle \varphi\gamma : \kappa'\rangle) \vdash$$
$$\mathcal{R}|c| : R\langle c : \kappa'\rangle$$

As we can easily show $R_{\mathsf{con}}(\Delta) \vdash \mu\chi.\kappa$ and $R_{\mathsf{con}}(\Delta) \vdash \lambda\delta{:}\mu\chi.\kappa.R\langle \delta : \mu\chi.\kappa\rangle : \mu\chi.\kappa \to \star$ we can derive by a simultaneous kind and constructor substitution for χ and φ_χ:

$$R_{\mathsf{con}}(\Delta), \alpha{:}\kappa[\mu\chi.\kappa/\chi], \varphi{:}\mu\chi.\kappa \to \kappa[\mu\chi.\kappa/\chi]';$$
$$R_{\mathsf{val}}(\Delta), x_\alpha{:}R\langle \alpha : \kappa[\mu\chi.\kappa/\chi]\rangle,$$
$$x_\varphi{:}\forall\gamma{:}\mu\chi.\kappa.(\lambda\delta{:}\mu\chi.\kappa.R\langle \delta : \mu\chi.\kappa\rangle)\gamma \to R\langle \varphi\gamma : \kappa'[\mu\chi.\kappa/\chi]\rangle$$
$$\vdash \mathcal{R}|c|[\mu\chi.\kappa/\chi, \lambda\delta{:}(\mu\chi.\kappa).R\langle \delta : \mu\chi.\kappa\rangle\varphi_\chi]$$
$$: R\langle c : \kappa'\rangle[\mu\chi.\kappa, \lambda\delta{:}\mu\chi.\kappa.R\langle \delta : \mu\chi.\kappa\rangle/\chi, \varphi_\chi]$$

By the lemma, we may note that result type of the judgment is actually

$$R\langle c[\mu\chi.\kappa/\chi] : \kappa'[\mu\chi.\kappa/\chi]\rangle.$$

We can easily show also that:

$$R_{\mathsf{con}}(\Delta), \beta{:}\mu\chi.\kappa \vdash \mathit{unfold}\, \beta : \kappa[\mu\chi.\kappa/\chi]$$
$$R_{\mathsf{con}}(\Delta) \vdash \rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]$$

With these facts we can apply the constructor substitution lemma again (after weakening the context with β) to substitute for α and φ:

$$R_{\mathsf{con}}(\Delta), \beta{:}\mu\chi.\kappa;$$
$$R_{\mathsf{val}}(\Delta), x_\alpha{:}R\langle \mathit{unfold}\, \beta : \kappa[\mu\chi.\kappa/\chi]\rangle,$$
$$x_\varphi{:}\forall\gamma{:}\mu\chi.\kappa.(\lambda\delta{:}\mu\chi.\kappa.R\langle \delta : \mu\chi.\kappa\rangle)\gamma \to R\langle \rho\gamma : \kappa'[\mu\chi.\kappa/\chi]\rangle \vdash$$
$$\mathcal{R}|c|[\mu\chi.\kappa/\chi, (\lambda\gamma{:}\mu\chi.\kappa.R\langle \gamma : \mu\chi.\kappa\rangle)/\varphi_\chi, \mathit{unfold}\, \beta/\alpha, \rho/\varphi] :$$
$$R\langle c[\mu\chi.\kappa, \mathit{unfold}\, \beta, \rho/\chi, \alpha, \varphi] : \kappa'[\mu\chi.\kappa/\chi]\rangle$$

To finish, we just need to notice two things—the type of x_φ is just $R\langle\rho : \mu\chi.\kappa \to \kappa'[\mu\chi.\kappa/\chi]\rangle$, what we wanted.

Furthermore,

$$R\langle c[\mu\chi.\kappa/\chi, \mathit{unfold}\,\beta/\alpha, \rho/\varphi] : \kappa'[\mu\chi.\kappa/\chi]\rangle$$

is equivalent to $R\langle\rho\beta : \kappa'[\mu\chi.\kappa/\chi\rangle$.

□

4.4.2 Encoding of LIR

$$\begin{array}{ccc} LI & \xrightarrow{\text{Informally, Section 4.3}} & LX \\ \text{Section 3.4}\downarrow & & \downarrow\text{Section 4.4} \\ LIR & \xrightarrow{\text{This section}} & LXR \end{array}$$

This section demonstrates that LXR is as fully expressive as LIR, by defining an embedding of LIR. This embedding consists of four translations: $|\cdot|_\kappa$ for kinds, $|\cdot|_c$ for constructors, $|\cdot|_t$ for types and $|\cdot|_e$ for terms. The techniques of this section are reminiscent of Section 4.3 in the embedding of kinds and constructors.

First we define the kind of the representation of LIR constructors:

$$T' = \mu\chi.(1 + (\chi \times \chi) + (\chi \times \chi))$$

Again for notational convenience, we will use $T'[k] = (1 + (\chi \times \chi) + (\chi \times \chi))[k/\chi]$ be its unrolling.

We use this definition as the base case for the embedding of LIR kinds into LX. Because T' is well formed, all kinds produced by this translation will be well formed.

$$\begin{aligned} |\star|_\kappa &= T' \\ |\kappa_1 \to \kappa_2|_\kappa &= |\kappa_1|_\kappa \to |\kappa_2|_\kappa \end{aligned}$$

Constructors in LIR are used in two ways—they are examined at the constructor level with *Typerec*, or they are interpreted as types. In the first case, we need to translate all LIR constructors into the constructor-level LX datatype. The translation of *Typerec* uses primitive recursion to create a fold over this datatype.

$$
\begin{array}{rl}
|\alpha|_c = & \alpha \\
|\lambda\alpha{:}\kappa.c|_c = & \lambda\alpha{:}|\kappa|_\kappa.|c|_c \\
|c_1 c_2|_c = & |c_1|_c|c_2|_c \\
|\,\mathit{int}\,|_c = & \mathit{fold}_{T'}(\mathit{inj}_1^{T'[T']} *) \\
|\rightarrow|_c = & \lambda\alpha{:}T'.\ \lambda\beta{:}T'.\ \mathit{fold}_{T'}(\mathit{inj}_2^{T'[T']}\langle\alpha,\beta\rangle) \\
|\times|_c = & \lambda\alpha{:}T'.\ \lambda\beta{:}T'.\ \mathit{fold}_{T'}(\mathit{inj}_3^{T'[T']}\langle\alpha,\beta\rangle) \\
|\,\mathit{Typerec}(c, c_{\mathit{int}}, c_{\rightarrow}, c_{\times})|_c = & \mathit{pr}_{T',\kappa}(\chi,\alpha,\varphi. \\
& \quad \mathit{case}\,\alpha, \\
& \quad\quad \mathit{inj}_1\,\beta \Rightarrow |\,c_{\mathit{int}}\,|_c, \\
& \quad\quad \mathit{inj}_2\,\beta \Rightarrow \mathit{case}\,\beta \\
& \quad\quad\quad \mathit{inj}_1\,\gamma \Rightarrow |\,c_{\rightarrow}\,|_c(\pi_1\gamma)(\pi_2\gamma) \\
& \quad\quad\quad\quad (\varphi(\pi_1\gamma))(\varphi(\pi_2\gamma)) \\
& \quad\quad\quad \mathit{inj}_2\,\gamma \Rightarrow |\,c_{\rightarrow}\,|_c(\pi_1\gamma)(\pi_2\gamma) \\
& \quad\quad\quad\quad (\varphi(\pi_1\gamma))(\varphi(\pi_2\gamma))) \\
& \quad |c|_c
\end{array}
$$

Next, when constructors are used as types in LIR, they need to be interpreted, again through primitive recursion. We use this interpretation as the basis for the embedding of types.

$$
\begin{array}{rl}
|\,T(c)|_t = & \mathit{pr}_{T'\star}(\chi,\alpha,\varphi. \\
& \quad \mathit{case}\,\alpha \\
& \quad\quad \mathit{inj}_1\,\beta \Rightarrow \mathit{int} \\
& \quad\quad \mathit{inj}_2\,\beta \Rightarrow \mathit{case}\,\beta \\
& \quad\quad\quad \mathit{inj}_1\,\gamma \Rightarrow (\varphi(\pi_1\gamma)) \rightarrow (\varphi(\pi_2\gamma)) \\
& \quad\quad\quad \mathit{inj}_2\,\gamma \Rightarrow (\varphi(\pi_1\gamma)) \times (\varphi(\pi_2\gamma))) \\
& \quad |c|_c \\
|t_1 \rightarrow t_2|_t = & |t_1|_t \rightarrow |t_2|_t \\
|t_1 \times t_2|_t = & |t_1|_t \times |t_2|_t \\
|\forall\alpha{:}\kappa.t|_t = & \forall\alpha{:}|\kappa|_\kappa.|t|_t \\
|\exists\alpha{:}\kappa.t|_t = & \exists\alpha{:}|\kappa|_\kappa.|t|_t
\end{array}
$$

Term representations already exist primitively in LIR, and we can encode them in the term level of LX as the encoding of the constructor representations. If the LIR type of a representation is $R(c)$ (it represents the LIR constructor c), then the LX type of the term representation should be $R\langle c : T'\rangle$. From the definition in Table 4.9, this should be a recursive type, parameterized by the translation of c. If we unfold the definitions, the type is

$$\mu_{T'}(\varphi, |c|_c)$$

where

$$\begin{array}{rl}\varphi & = \lambda\alpha_\chi{:}T' \to \star.\lambda\beta{:}T'.R\langle \mathit{unfold}\,\beta : T'[\chi]\rangle \\ & = \lambda\alpha_\chi{:}T' \to \star.\lambda\beta{:}T'.\mathit{case}(\mathit{unfold}\,\beta, \mathit{inj}_1\,\alpha \Rightarrow \mathit{unit}, \mathit{inj}_2\,\alpha \Rightarrow \mathit{void}) \\ & \quad + \mathit{case}(\mathit{unfold}\,\beta, \\ & \quad\quad \mathit{inj}_1\,\alpha \Rightarrow \mathit{void}, \\ & \quad\quad \mathit{inj}_2\,\alpha \Rightarrow \mathit{case}\,\alpha \\ & \quad\quad\quad \mathit{inj}_1\,\gamma \Rightarrow \alpha_\chi(\pi_1\gamma) \times \alpha_\chi(\pi_2\gamma) \\ & \quad\quad\quad \mathit{inj}_2\,\gamma \Rightarrow \mathit{void}) \\ & \quad + \mathit{case}(\mathit{unfold}\,\beta, \\ & \quad\quad \mathit{inj}_1\,\alpha \Rightarrow \mathit{void}, \\ & \quad\quad \mathit{inj}_2\,\alpha \Rightarrow \mathit{case}\,\alpha \\ & \quad\quad\quad \mathit{inj}_1\,\gamma \Rightarrow \mathit{void} \\ & \quad\quad\quad \mathit{inj}_2\,\gamma \Rightarrow \alpha_\chi(\pi_1\gamma) \times \alpha_\chi(\pi_2\gamma))\end{array}$$

The definitions of the term representations follow from Table 4.10.

$$\begin{array}{rcl}|\,R_{\mathit{int}}\,|_e = \mathcal{R}||\,\mathit{int}\,|_c| & = & \mathit{fold}_{\mu_{T'}(\varphi,|c|_c)}(\mathit{inj}_1()) \\ |\,R_{\to}\,|_e = \mathcal{R}||\to|_c| & = & \Lambda\alpha{:}T'.\lambda x_\alpha{:}R\langle\alpha : T'\rangle.\ \Lambda\beta{:}T.\lambda x_\beta{:}R\langle\beta : T'\rangle. \\ & & \quad \mathit{fold}_{\mu_{T'}(\varphi,|c|_c)}(\mathit{inj}_2\langle x_\alpha, x_\beta\rangle) \\ |\,R_{\times}\,|_e = \mathcal{R}||\times|_c| & = & \Lambda\alpha{:}T'.\lambda x_\alpha{:}R\langle\alpha : T'\rangle.\ \Lambda\beta{:}T.\lambda x_\beta{:}R\langle\beta : T'\rangle. \\ & & \quad \mathit{fold}_{\mu_{T'}(\varphi,|c|_c)}(\mathit{inj}_3\langle x_\alpha, x_\beta\rangle)\end{array}$$

To analyze these term representations, we use a combination of *case* and *vcase*. If the representation argument e to a *typerec* term is of LIR type $R(c)$, and the entire *typerec* term is of type $\sigma[c/\delta]$, then we can embed it in LXR as below.

$$
\begin{array}{l}
|\,\mathit{typerec}[\delta.\sigma](e, e_{int}, e_{\to}, e_{\times})|_e = \\
\\
(\mathit{fix}\ f{:}\forall\alpha{:}\star.R(\alpha) \to \sigma[\alpha/\delta]. \\
\quad \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\alpha). \\
\quad\quad \mathit{case}(\mathit{unfold}\, x_\alpha) \\
\quad\quad\quad \mathit{inj}_1\, x \Rightarrow \mathit{vcase}(\mathit{unfold}\, c) \\
\quad\quad\quad\quad \mathit{inj}_1\, \beta \Rightarrow |\, e_{int}\,|_e \\
\quad\quad\quad\quad \mathit{inj}_2\, \beta \Rightarrow \mathit{dead}\, x \\
\quad\quad\quad \mathit{inj}_2\, x \Rightarrow \mathit{vcase}(\mathit{unfold}\, c) \\
\quad\quad\quad\quad \mathit{inj}_1\, \beta \Rightarrow \mathit{dead}\, x \\
\quad\quad\quad\quad \mathit{inj}_2\, \beta \Rightarrow \mathit{case}\ x\ \mathit{of} \\
\quad\quad\quad\quad\quad \mathit{inj}_1\, y \Rightarrow (\mathit{vcase}\, \beta \\
\quad\quad\quad\quad\quad\quad \mathit{inj}_1\, \gamma \Rightarrow \mathit{let}\langle w, z\rangle = y\ \mathit{in} \\
\quad\quad\quad\quad\quad\quad\quad |\, e_{\to}\,|_e[\pi_1\gamma]\ w\ (f[\pi_1\gamma]w)\ [\pi_2\gamma]\ z\ (f[\pi_2\gamma]z) \\
\quad\quad\quad\quad\quad\quad \mathit{inj}_2\, \gamma \Rightarrow \mathit{dead}\, y) \\
\quad\quad\quad\quad\quad \mathit{inj}_2\, y \Rightarrow (\mathit{vcase}\, \beta \\
\quad\quad\quad\quad\quad\quad \mathit{inj}_1\, \gamma \Rightarrow \mathit{dead}\, y \\
\quad\quad\quad\quad\quad\quad \mathit{inj}_2\, \gamma \Rightarrow \mathit{let}\langle w, z\rangle = x\ \mathit{in} \\
\quad\quad\quad\quad\quad\quad\quad |\, e_{\times}\,|_e[\pi_1\gamma]\ w\ (f[\pi_1\gamma]w)\ [\pi_2\gamma]\ z\ (f[\pi_2\gamma]z)) \\
\quad)\ [|c|_c]\ |e|_e
\end{array}
$$

Finally, the rest of the term embedding can be defined in a straightforward manner.

$$
\begin{array}{rl}
|x|_e = & x \\
|\lambda x{:}\sigma.e|_e = & \lambda x{:}|\sigma|_t.|e|_e \\
|\,\mathit{fix}\ f{:}\sigma.v|_e = & \mathit{fix}\ f{:}|\sigma|_t.|v|_e \\
|e_1 e_2|_e = & |e_1|_e|e_2|_e \\
|\langle e_1, e_2\rangle|_e = & \langle |e_1|_e, |e_2|_e\rangle \\
|\pi_1 e|_e = & \pi_1|e|_e \\
|\pi_2 e|_e = & \pi_2|e|_e \\
|\Lambda\alpha{:}\kappa.v|_e = & \Lambda\alpha{:}|\kappa|_\kappa.|v|_e
\end{array}
$$

Definition 4.4.4 *Define the translation of contexts in a pointwise manner:*

$$
\begin{array}{lcl}
|\emptyset| & = & \emptyset \\
|\Delta, \alpha : \kappa| & = & |\Delta|, \alpha{:}|\kappa|_\kappa \\
\\
|\emptyset| & = & \emptyset \\
|\Gamma, x : \sigma| & = & |\Gamma|, x : |\sigma|_t
\end{array}
$$

Theorem 4.4.5 (Static correctness of translation) *1. For all κ in LIR, $\emptyset \vdash |\kappa|_\kappa$.*

2. If $\Delta; \Gamma \vdash_R c : \kappa$ then $|\Delta|; |\Gamma| \vdash |c|_c : |\kappa|_\kappa$.

3. If $\Delta; \Gamma \vdash_R e : \sigma$ then $|\Delta|; |\Gamma| \vdash |e|_e : |\sigma|_t$.

4.5 Discussion and chapter summary

In this chapter, I have described a language designed to encode several type systems simultaneously. Through the use of an expressive programming language at the type level, the type structure of various intermediate languages and the translations between them may be described and reasoned about within LX.

This chapter is important to the rest of the thesis for several additional reasons. First, it raises the question of what linguistic support is actually necessary to support intensional type analysis. In essence, the only specialized constructs in LX and in LXR are those for constructor refinement. In the next chapter, I will show that even those terms are unnecessary, and use impredicative polymorphism to encode LIR.

Furthermore, this chapter presents the first solution to the problem of analyzing types with binding structure. By representing these types with abstract syntax, and then analyzing that abstract syntax, LX may express type analysis over the types of the polymorphic lambda calculus. This facility is important for advanced type systems that include such types in an intrinsic way. In chapter 6, I will come back to this issue, and present a different solution for analyzing polymorphic types.

Finally, the mechanism used in this chapter for creating the representations of constructors and describing their types is interesting in its own respect. In essence, it is creating a form of synthetic dependent type. By using similar techniques, Crary and Weirich [CW00] developed a language that could express type dependency on term values. This language, called LXres, used this expressiveness to describe the running time of programs. Because this running time of a function typically depends on the arguments to that function, this language could use that dependency to describe that relationship, and so was quite expressive. Most importantly, because LXres was not based on a full dependent type system, it retained decidable type checking. As in LXR, there was a complete separation between the type language with a decidable equivalence procedure and possibly non-terminating term language.

Chapter 5

Type analysis without hard-wired types (II)

5.1 Eliminating type analysis

In this chapter, I further explore what is necessary to implement run-time type analysis. In fact, little hard-wired machinery is necessary for expressing programs that analyze types. In other words, instead of using the sums, products and primitive recursion in the type language of LXR and *vcase* in the term language of LXR, in both cases we can use impredicative polymorphism to encode the type language of LIR. Because types essentially form an inductive datatype (that is, the kind $\star$ is an inductive datatype with data constructors *int*, $\rightarrow$, etc.), we can use well-known techniques of encoding such datatypes. The fact that our target language contains nothing beyond impredicative polymorphism further justifies the claim that intensional type analysis is simply a *programming idiom* that is possible in a sufficiently rich language.

This chapter essentially describes an embedding of LIR into a pared down language called LU. I begin this chapter with an informal description of the technique that we use for that embedding. Formal descriptions of the minor changes we must make to LIR and of the target languages LU appear in Section 5.2, and I present the embedding between them in Section 5.3. Section 5.4 describes the limitations of the translation and discusses when one might want an explicit iteration operator in the target language, such as *pr* in the last chapter.

5.1.1 Encoding datatypes with polymorphism

Consider a well-known inductive datatype (presented in Standard ML syntax [MTHM97] augmented with explicit polymorphism):

```
datatype Tree = Leaf | Node of Tree * Tree
Treerec : ∀a. Tree -> a -> ( a * a -> a ) -> a
```

`Leaf` and `Node` are introduction forms, used to create elements of type `Tree`. The function `Treerec` is an elimination form, iterating computation over an element of type `Tree`, creating a fold or a catamorphism. It accepts a base case (of type `a`) for the leaves and an inductive case (of type `a * a -> a`) for the nodes . For example, we may use `Treerec` to define a function to display a `Tree`. First, we explicitly instantiate the return type `a` with `[string]`. For the leaves, we provide the string `"Leaf"`, and for the nodes we concatenate (with the infix operator ^) the strings of the subtrees.

```
val showTree = fn x : Tree =>
  Treerec [string] x
     "Leaf"
     (fn (s1:string, s2:string) => "Node(" ++ s1 ++ "," ++ s2 ")")
```

As `Tree` is an inductive datatype, it is well known how to encode it in the polymorphic lambda calculus [BB85]. The basic idea is to encode a `Tree` as its elimination form—a function that iterates over the tree. In other words, a `Leaf` is a function that accepts a base case and an inductive case and returns the base case. Because we do not wish to constrain the return type of iteration, we abstract it, explicitly binding it with Λ`a`.

```
val Leaf = Λa. fn base:a => fn ind:a * a -> a => base
```

Likewise, a `Node`, with two subtrees `x` and `y`, selects the inductive case, passing it the result of continuing the iteration through the two subtrees.

```
val Node (x:Tree) (y:Tree) =
   Λa. fn base:a  => fn ind:a * a -> a =>
       ind (Treerec [a] x base ind) (Treerec [a] y base ind)
```

However, as all of the iteration is encoded into the data structure itself, the elimination form only needs to pass it on.

```
val Treerec = Λa. fn x : Tree => fn base : a =>
              fn ind  : a * a -> a => x [a] base ind
```

Consequently, we may write `Node` more simply as

```
val Node (x:Tree) (y:Tree) =
   Λa. fn base:a  => fn ind:a * a -> a =>
       ind (x [a] base ind) (y [a] base ind)
```

Now consider another inductive datatype:

```
datatype Type = Int | Arrow of Type * Type
Typerec : ∀a. Type -> a -> ( a * a -> a ) -> a
```

Ok, so I just changed the names. But this encoding will still provide us with methodology for encoding a type analyzing language such as LIR, into a language only containing polymorphism. As there are two elimination forms for LIR (*typerec* that eliminates representations to terms and *Typerec* that eliminates type constructors to types) this encoding occurs at two different levels. Therefore our simplest target language is Girard's LU, the language F_ω augmented with kind polymorphism at the type constructor level (Table 5.2).

5.2 Source and target language details

Although in previous chapters *typerec* and *Typerec* define a primitive recursive fold over kind $\star$ (also called a *paramorphism* [MFP91, Mee92]), in this modification of LIR we replace these operators with their iterative cousins (which define *catamorphisms*). The difference between iteration and primitive recursion is apparent in the kind of $c_\rightarrow$ and the type of $e_\rightarrow$. With primitive recursion, the arrow branch receives four arguments: the two subcomponents of the arrow constructor and two results of continuing the fold through these subcomponents. In iteration, on the other hand, the arrow branch receives only two arguments, the results of the continued fold.[1] We discuss this restriction further in Section 5.4.2.

Furthermore, in order to simplify the proof of dynamic correctness of the translation into LU, there is one more modification. Instead of giving LIR a call-by-value semantics, for this chapter we consider a version of it with a call-by-name semantics. This change means that instead of the restricted β-evaluation rule that only applies when its argument is a value:

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

we allow the full rule for function application:

$$(\lambda x{:}\sigma.e)e' \mapsto_R e[e'/x]$$

[1] Because we cannot separate type constructors passed for static type checking, from those passed for dynamic type analysis in LI, we *must* provide the subcomponents of the arrow type to the arrow branch of *typerec*. Therefore, we cannot define an iterative version of *typerec* for that language.

Table 5.1: *Typerec*^{it} and *typerec*^{it}

$$\frac{\Delta \vdash \tau : \star \quad \Delta \vdash c_{int} : \kappa \quad \Delta \vdash c_{\rightarrow} : \kappa \rightarrow \kappa \rightarrow \kappa}{\Delta \vdash \mathit{Typerec}^{it}[\kappa]\ \tau\ \ c_{int}\ \ c_{\rightarrow}\ \ : \kappa}$$

$$\frac{}{\Delta \vdash \mathit{Typerec}^{it}[\kappa]\ \mathit{int}\ \ c_{int}\ \ c_{\rightarrow} = c_{int} : \kappa}$$

$$\frac{}{\begin{array}{l}\Delta \vdash \mathit{Typerec}^{it}[\kappa](\tau_1 \rightarrow \tau_2)\ \ c_{int}\ \ c_{\rightarrow} \\ \quad = c_{\rightarrow}\ (\mathit{Typerec}^{it}[\kappa]\ \tau_1\ \ c_{int}\ \ c_{\rightarrow})(\mathit{Typerec}^{it}[\kappa]\ \tau_2\ \ c_{int}\ \ c_{\rightarrow}) : k\end{array}}$$

$$\frac{\Delta, \alpha{:}\star \vdash \sigma \quad \Delta; \Gamma \vdash e : R(c) \quad \Delta; \Gamma \vdash e_{int} : \sigma[\mathit{int}/\alpha] \quad \Delta; \Gamma \vdash e_{\rightarrow} : \forall\beta{:}\star.\sigma[\beta/\alpha] \rightarrow \forall\gamma{:}\star.\sigma[\gamma/\alpha] \rightarrow \sigma[(\beta \rightarrow \gamma)/\alpha]}{\Delta; \Gamma \vdash \mathit{typerec}^{it}[\alpha.\sigma]\ e\ \ e_{int}\ \ e_{\rightarrow} : \sigma[c/\alpha]}$$

$$\frac{}{\mathit{typerec}^{it}[\alpha.\sigma]\ \ R_{int}\ \ e_{int}\ \ e_{\rightarrow} \mapsto e_{int}}$$

$$\frac{}{\begin{array}{l}\mathit{typerec}^{it}[\alpha.\sigma]\ \ (R_{\rightarrow}[\tau_1]e_1\ \ [\tau_2]\ e_2)\ \ e_{int}\ \ e_{\rightarrow} \\ \quad \mapsto e_{\rightarrow}\ \ [\tau_1]\ \ (\mathit{typerec}^{it}[\alpha.\sigma]\ e_1\ \ e_{int}\ \ e_{\rightarrow})[\tau_2]\ \ (\mathit{typerec}^{it}[\alpha.\sigma]\ e_2\ \ e_{int}\ \ e_{\rightarrow})\end{array}}$$

In order to maintain a deterministic semantics, we must remove the following rule from LIR that allows us to evaluate the function argument.

$$\frac{e \mapsto_R e'}{ve \mapsto_R ve'}$$

Table 5.2: LU: Syntax

(*kinds*)	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2 \mid \chi \mid \forall\chi.\kappa$
(*con's*)	c, τ	$::=$	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \Lambda\chi.c \mid c[\kappa]$ $\mid int \mid \tau_1 \to \tau_2 \mid \forall\alpha{:}\kappa.\tau$
(*terms*)	e	$::=$	$i \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2$ $\mid \Lambda\alpha{:}\kappa.e \mid e[c]$

Table 5.3: LU: Operational semantics

$$(\lambda x{:}c.e)e' \mapsto e[e'/x]$$

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}$$

$$(\Lambda\alpha{:}\kappa.e)[c] \mapsto (e[c/\alpha])$$

$$\frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

Table 5.4: LU: Static semantics

$\boxed{\Delta \vdash \kappa}$ **Kind Formation**

[*k-var*]

$$\frac{}{\Delta, \chi \vdash \chi}$$

[*k-type*]

$$\frac{}{\Delta \vdash \star}$$

Table 5.4 (Continued)

[*k-arr*]	$$\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 \to \kappa_2}$$
[*k-all*]	$$\frac{\Delta, \chi \vdash \kappa}{\Delta \vdash \forall \chi.\kappa} \; (\chi \notin \Delta)$$
$\boxed{\Delta \vdash c : \kappa}$	**Constructor Formation**
[*c-var*]	$$\frac{}{\Delta \vdash \alpha : \kappa} \; (\Delta(\alpha) = \kappa)$$
[*c-fn*]	$$\frac{\Delta, \alpha{:}\kappa_1 \vdash c : \kappa_2}{\Delta \vdash \lambda \alpha{:}\kappa_1.c : \kappa_1 \to \kappa_2} \; (\alpha \notin Dom(\Delta))$$
[*c-app*]	$$\frac{\Delta \vdash c_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash c_2 : \kappa_1}{\Delta \vdash c_1 c_2 : \kappa_2}$$
[*c-kfn*]	$$\frac{\Delta, \chi \vdash c : \kappa}{\Delta \vdash \Lambda \chi.c : \forall \chi.\kappa} \; (\chi \notin \Delta)$$
[*c-kapp*]	$$\frac{\Delta \vdash c : \forall \chi.\kappa}{\Delta \vdash c[\kappa_1] : \kappa[\kappa_1/\chi]}$$
[*c-int-type*]	$$\frac{}{\Delta \vdash int : \star}$$
[*c-fn-type*]	$$\frac{\Delta \vdash c_1 : \star \qquad \Delta \vdash c_2 : \star}{\Delta \vdash c_1 \to c_2 : \star}$$
[*c-all-type*]	$$\frac{\Delta, \alpha{:}\kappa \vdash c : \star \qquad \alpha \notin Dom(\Delta)}{\Delta \vdash \forall \alpha{:}\kappa.c : \star}$$

Table 5.4 (Continued)

$\Delta \vdash c = c' : \kappa$	**Constructor Equality**
[*ceq-β*]	$$\frac{\Delta, \alpha{:}\kappa' \vdash c_1 : \kappa \qquad \Delta \vdash c_2 : \kappa'}{\Delta \vdash (\lambda\alpha{:}\kappa'.c_1)c_2 = c_1[c_2/\alpha] : \kappa} \ (\alpha \notin Dom(\Delta))$$
[*ceq-η*]	$$\frac{\Delta \vdash c : \kappa_1 \to \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.c\,\alpha = c : \kappa_1 \to \kappa_2} \ (\alpha \notin Dom(\Delta))$$
[*ceq-cong1*]	$$\frac{\Delta, \alpha{:}\kappa \vdash c = c' : \kappa'}{\Delta \vdash \lambda\alpha{:}\kappa.c = \lambda\alpha{:}\kappa.c' : \kappa \to \kappa'}$$
[*ceq-cong2*]	$$\frac{\Delta \vdash c_1 = c_1' : \kappa' \to \kappa \quad \Delta \vdash c_2 = c_2' : \kappa'}{\Delta \vdash c_1 c_2 = c_1' c_2' : \kappa}$$
[*ceq-κβ*]	$$\frac{\Delta, \chi \vdash c : \kappa'}{\Delta \vdash \Lambda\chi.c[\kappa] = c[\kappa/\chi] : \kappa'[\kappa/\chi]} \ (\chi \notin \Delta)$$
[*ceq-κη*]	$$\frac{\Delta \vdash c : \forall\chi'.\kappa}{\Delta \vdash \Lambda\chi.c[\chi] = c : \forall\chi'.\kappa}$$
[*ceq-cong3*]	$$\frac{\Delta, \chi \vdash c = c' : \kappa}{\Delta \vdash \Lambda\chi.c = \Lambda\chi.c' : \forall\chi.\kappa} \ (\chi \notin \Delta)$$
[*ceq-cong4*]	$$\frac{\Delta \vdash c = c' : \forall\chi.\kappa}{\Delta \vdash c[\kappa] = c'[\kappa] : \kappa'[\kappa/\chi]}$$
[*ceq-cong5*]	$$\frac{\Delta \vdash c_1 = c_1' : \star \quad \Delta \vdash c_2 = c_2' : \star}{\Delta \vdash c_1 \to c_2 = c_1' \to c_2' : \star}$$

Table 5.4 (Continued)

[*ceq-cong6*]	$$\frac{\Delta, \alpha{:}\kappa \vdash \tau = \tau' : \star}{\Delta \vdash \forall \alpha{:}\kappa.\tau = \forall \alpha{:}\kappa.\tau' : \star}$$
[*ceq-ref*]	$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa}$$
[*ceq-sym*]	$$\frac{\Delta \vdash c' = c : \kappa}{\Delta \vdash c = c' : \kappa}$$
[*ceq-trans*]	$$\frac{\Delta \vdash c = c' : \kappa \qquad \Delta \vdash c' = c'' : \kappa}{\Delta \vdash c = c'' : \kappa}$$
$\boxed{\Delta; \Gamma \vdash e : \tau}$	**Term Formation**
[*e-int*]	$$\frac{}{\Delta; \Gamma \vdash i : int}$$
[*e-var*]	$$\frac{}{\Delta; \Gamma \vdash x : \tau} \ (\Gamma(x) = \tau)$$
[*e-fn*]	$$\frac{\Delta; \Gamma, x{:}\tau_2 \vdash e : \tau_1 \qquad \Delta \vdash \tau_2 : \star}{\Delta; \Gamma \vdash \lambda x{:}\tau_2.e : \tau_2 \to \tau_1} \ (x \notin Dom(\Gamma))$$
[*e-app*]	$$\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1 e_2 : \tau_1}$$
[*e-tfn*]	$$\frac{\Delta; \Gamma \vdash e : \forall \alpha{:}\kappa.\tau \quad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash e[c] : \tau[c/\alpha]}$$
[*e-tapp*]	$$\frac{\Delta, \alpha{:}\kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.\tau} \ (\alpha \notin Dom(\Delta))$$
[*e-equiv*]	$$\frac{\Delta; \Gamma \vdash e : \tau_2 \quad \Delta \vdash \tau_1 = \tau_2 : \star}{\Delta; \Gamma \vdash e : \tau_1}$$

The target language of the translation is LU, originally called "System $U-$" by Girard [Gir72]. Essentially it is the language F_ω augmented with kind polymorphism at the type constructor level (Table 5.2). We make the type constructor language polymorphic by adding kind variables χ and polymorphic kinds $\forall\chi.\kappa$ to the syntax of kinds, and adding type constructors supporting kind abstraction $(\Lambda\chi.c)$ and application $c[\kappa]$. Otherwise, the constructor language resembles that of a standard polymorphic lambda calculus. As the target language is impredicative, both types and type constructors are in the same syntactic class[2].

The dynamic and static semantics of LU appear in Tables 5.3 and 5.4. This semantics is very similar to that of the core language of Chapter 2. For consistency, LU is presented with a small-step call-by-name operational semantics. The notation $\mapsto_U$ emphasizes that the rules apply to LU, when it is not clear from context. We also use $\vdash_U$ to differentiate the typing judgments of LU. This static semantics must make sure that kind variables appear correctly. As in the last chapter, we add kind variables to the context Δ to type check constructors and kinds. However, this time there are no restrictions that the kind variables appear positively. The rules of the static semantics that specifically support kind polymorphism are shaded in the tables.

5.3 Defining iteration

The translation of LIR into LU can be thought of as two separate translations: A translation of the kinds and constructors of LIR into the kinds and constructors of LU and a translation of the types and terms of LIR into the constructors and terms of LU. For reference, the complete translation appears in Tables 5.5 and 5.6.

To define the translation of $\mathit{Typerec}^{it}$ we use the traditional encoding of inductive datatypes in impredicative polymorphism. As before, we encode τ, of kind $\star$ as its elimination form: a function that chooses between two given branches—one for c_{int}, one for $c_{\to}$. Then $\mathit{Typerec}^{it}[\kappa]\ \tau\ \ c_{int}\ \ c_{\to}$ can be implemented with

$$|\tau|[|\kappa|]\ |\ c_{int}\ |\ |\ c_{\to}\ |$$

As τ is of kind type, we define $|\star|$ to reflect the fact that $|\tau|$ must accept an arbitrary kind and the two branches.

$$|\star| = \forall\chi.\chi \to (\chi \to \chi \to \chi) \to \chi$$

[2]In Section 5.4.3 we discuss why we might want alternate target languages not based on impredicative polymorphism.

Accordingly, the encoding of the type constructor *int* just returns its first argument (the kinds of the arguments have been elided)

$$|\,int\,| = (\Lambda\chi.\lambda\iota.\lambda\alpha.\iota)$$

Now consider the constructor equality rule when the argument to $Typerec^{it}$ is an arrow type. The translation of the arrow type constructor $\rightarrow$, should apply the second argument (the $c_{\rightarrow}$ branch) to the result of continuing the recursion through the two subcomponents.

$$|\tau_1 \rightarrow \tau_2| = \Lambda\chi.\lambda\iota.\lambda\alpha.\alpha(|\tau_1|[\chi]\ \iota\ \alpha)(|\tau_2|[\chi]\ \iota\ \alpha)$$

A critical property of this translation is that it preserves the equivalences that exist in the source language. For example, one equivalence we must preserve from the source language is that

$$\begin{array}{l} |\,Typerec^{it}[\kappa]\ (\tau_1 \rightarrow \tau_2)\ \ c_{int}\ \ c_{\rightarrow}\,| \\ \quad = |\,c_{\rightarrow}(Typerec^{it}[\kappa]\ \tau_1\ \ c_{int}\ \ c_{\rightarrow})(Typerec^{it}[\kappa]\ \tau_2\ \ c_{int}\ \ c_{\rightarrow})| \end{array}$$

If we expand the left side, we get

$$(\Lambda\chi.\lambda\iota.\lambda\alpha.\alpha(|\tau_1|[\chi]\ \iota\ \alpha)(|\tau_2|[\chi]\ \iota\ \alpha))\ [|\kappa|]\ |\,c_{int}\,|\ |\,c_{\rightarrow}\,|$$

This term is then β-equivalent to the expansion of the right hand side.

$$|\,c_{\rightarrow}\,|\ (|\tau_1|[|\kappa|]|\,c_{int}\,||\,c_{\rightarrow}\,|)\ (|\tau_2|[|\kappa|]|\,c_{int}\,||\,c_{\rightarrow}\,|)$$

Because type constructors are a separate syntactic class from types, we must define $|T(\tau)|$, the coercion between them. We convert $|\tau|$ of kind $|\star|$ into a LU constructor of kind $\star$ using the iteration built into $|\tau|$.

$$|T(\tau)| = |\tau|\ [\star]\ \ int\ (\lambda\alpha{:}\star.\lambda\beta{:}\star.\alpha \rightarrow \beta)$$

For example,

$$\begin{array}{ll} |T(int)| & = |\,int\,|[\star]\ \ int\ (\lambda\alpha{:}\star.\lambda\beta{:}\star.\alpha \rightarrow \beta) \\ & = (\Lambda\chi.\lambda\iota.\lambda\alpha.\iota)[\star]\ \ int\ (\lambda\alpha{:}\star.\lambda\beta{:}\star.\alpha \rightarrow \beta) \\ & =_\beta int \end{array}$$

We use a very similar encoding for $typerec^{it}$ at the term level, as we do for $Typerec^{it}$. Again, we wish to apply the translation of the argument to the translation of the branches, and let the argument select between them.

$$|\,typerec^{it}[\alpha.\sigma]\ e\ \ e_{int}\ \ e_{\rightarrow}\,| \quad \text{as} \quad |e|\ [\lambda\alpha{:}|\star|.|\sigma|]\ |\,e_{int}\,|\ |\,e_{\rightarrow}\,|$$

Table 5.5: Translation of LIR into LU, kinds and constructors

Kind Translation

$$
\begin{array}{lcl}
|\star| & = & \forall\chi.\chi \to (\chi \to \chi \to \chi) \to \chi \\
|\kappa_1 \to \kappa_2| & = & |\kappa_1| \to |\kappa_2|
\end{array}
$$

Constructor Translation

$$
\begin{array}{lcl}
|\alpha| & = & \alpha \\
|\lambda\alpha{:}\kappa.c| & = & \lambda\alpha{:}|\kappa|.|c| \\
|c_1 c_2| & = & |c_1||c_2| \\
|\mathit{int}| & = & \Lambda\chi.\lambda\iota{:}\chi.\lambda\alpha{:}\chi \to \chi \to \chi.\iota \\
|\to| & = & \lambda\alpha_1{:}|\star|.\lambda\alpha_2{:}|\star|.\Lambda\chi.\lambda\iota{:}\chi.\lambda\alpha{:}\chi \to \chi \to \chi. \\
 & & \quad \alpha\ (\alpha_1\ [\chi]\ \iota\ \alpha)\ (\alpha_2\ [\chi]\ \iota\ \alpha) \\
|\mathit{Typerec}^{it}[\kappa]\ \tau\ \ c_{int}\ \ c_{\to}| & = & |\tau|\ [|\kappa|]\ |c_{int}|\ |c_{\to}|
\end{array}
$$

The translations of R_{int} and $R_{\to}$ are analogous to those of the type constructors *int* and $\to$. To preserve typing, we define $|R\tau|$ as:

$$
\begin{array}{l}
\forall\gamma{:}|\star| \to \star.\ \gamma|\mathit{int}| \\
\quad \to (\forall\alpha{:}|\star|.\gamma\alpha \to \forall\beta{:}|\star|.\gamma\beta \to \gamma|\alpha \to \beta|) \\
\quad \to \gamma|\tau|
\end{array}
$$

5.3.1 Properties of the embedding

The translation presented above enjoys the following properties. Define $|\Delta|$ as $\{\alpha{:}|\Delta(\alpha)| \mid \alpha \in \mathit{Dom}(\Delta)\}$ and $|\Gamma|$ as $\{x{:}|\Gamma(x)| \mid x \in \mathit{Dom}(\Gamma)\}$.

First we show the correctness of the translation. The translation of terms that type check in the source language also type check in the target language.

Theorem 5.3.1 (Static Correctness) *1.* $\emptyset \vdash_U |\kappa|$

2. If $\Delta \vdash_R c : \kappa$ *then* $|\Delta| \vdash_U |c| : |\kappa|$.

3. If $\Delta \vdash_R c = c' : \kappa$ *then* $|\Delta| \vdash_U |c| = |c'| : |\kappa|$.

4. If $\Delta \vdash_R \sigma$ *then* $|\Delta| \vdash_U |\sigma| : \star$.

Table 5.6: Translation of LIR into LU, types and terms

Type Translation

$$
\begin{array}{lcl}
|\,T(\tau)| & = & |\tau|\ [\star]\ \ int\ \ \rightarrow \\
|R\tau| & = & \forall\gamma{:}|\star| \rightarrow \star.\ \gamma|\,int\,| \\
 & & \quad \rightarrow (\forall\alpha{:}|\star|.\gamma\alpha \rightarrow \forall\beta{:}|\star|.\gamma\beta \rightarrow \gamma|\alpha \rightarrow \beta|) \\
 & & \quad \rightarrow \gamma|\tau| \\
|\,int\,| & = & int \\
|\sigma_1 \rightarrow \sigma_2| & = & |\sigma_1| \rightarrow |\sigma_2| \\
|\forall\alpha{:}\kappa.\sigma| & = & \forall\alpha{:}|\kappa|.|\sigma|
\end{array}
$$

Term Translation

$$
\begin{array}{lcl}
|x| & = & x \\
|\lambda x{:}\sigma.e| & = & \lambda x{:}|\sigma|.|e| \\
|e_1 e_2| & = & |e_1||e_2| \\
|\Lambda\alpha{:}\kappa.e| & = & \Lambda\alpha{:}|\kappa|.|e| \\
|e[c]| & = & |e|[|c|] \\
|\,R_{int}\,| & = & (\Lambda\gamma{:}|\star| \rightarrow \star.\ \lambda i{:}\gamma|\,int\,|. \\
 & & \quad \lambda a{:}(\forall\alpha{:}|\star|.\gamma\alpha \rightarrow \forall\beta{:}|\star|.\gamma\beta \rightarrow \gamma|\alpha \rightarrow \beta|)\ .i) \\
|\,R_{\rightarrow}\,| & = & \Lambda\alpha{:}|\star|.\lambda x_1{:}|R\alpha|.\Lambda\beta{:}|\star|.\lambda x_2{:}|R\beta| \\
 & & \quad (\Lambda\gamma{:}|\star| \rightarrow \star.\ \lambda i{:}\gamma|\,int\,|. \\
 & & \quad\quad \lambda a{:}(\forall\alpha{:}|\star|.\gamma\alpha \rightarrow \forall\beta{:}|\star|.\gamma\beta \rightarrow \gamma|\alpha \rightarrow \beta|). \\
 & & \quad\quad\quad a\ [\alpha]\ (x_1[\gamma]\ i\ a)\ [\beta]\ (x_2[\gamma]\ i\ a)) \\
|\,typerec^{it}[\alpha.\sigma]\ e\ \ e_{int}\ \ e_{\rightarrow}\,| & = & |e|[\lambda\alpha{:}|\star|.|\sigma|]|\,e_{int}\,|\ |\,e_{\rightarrow}\,|
\end{array}
$$

5. *If* $\Delta \vdash_R \sigma = \sigma'$ *then* $|\Delta| \vdash_U |\sigma| = |\sigma'| : \star$

6. *If* $\Delta; \Gamma \vdash_R e : \sigma$ *then* $|\Delta; \Gamma| \vdash_U |e| : |\sigma|$.

Proof

Proof is by induction on the appropriate derivation. □

Furthermore, evaluation in the source language is mirrored by evaluation in the target language.

Theorem 5.3.2 (Dynamic Correctness) *If* $\emptyset \vdash_R e : \tau$ *and* $e \mapsto_R e'$ *then* $|e| \mapsto_U^* |e'|$.

Proof

The proof of this result is fairly straightforward, by induction on $e \mapsto e'$. The most complex case is the rule for *typerec* when the argument is a function type. Suppose

$$\begin{array}{l} \mathit{typerec}^{it}[\alpha.\sigma]\ (R_{\mathit{int}}[\tau_1]\ e_1\ [\tau_2]\ e_2)\ \ e_{\mathit{int}}\ \ e_{\rightarrow} \\ \quad \mapsto_R e_{\rightarrow}\ [\tau_1]\ (\mathit{typerec}^{it}[\alpha.\sigma]\ e_1\ \ e_{\mathit{int}}\ \ e_{\rightarrow})\ [\tau_2]\ (\mathit{typerec}^{it}[\alpha.\sigma]\ e_1\ \ e_{\mathit{int}}\ \ e_{\rightarrow}) \end{array}$$

the translation of the left side is the following:

$$\begin{array}{l} ((\Lambda\alpha{:}|\star|.\lambda x_1{:}|R\alpha|.\Lambda\beta{:}|\star|.\lambda x_2{:}|R\beta|. \\ \quad \Lambda\gamma{:}|\star| \rightarrow \star.\lambda i{:}(\gamma|\mathit{int}|).\lambda a{:}(\forall\alpha.\gamma\alpha \rightarrow \forall\beta.\gamma\beta \rightarrow \gamma|\alpha \rightarrow \beta|). \\ \quad\quad a\ [\alpha]\ (x_1[\gamma]\ i\ a)\ [\beta]\ (x_2[\gamma]\ i\ a)) \\ \quad [|\tau_1|]\ |e_1|\ [|\tau_2|]\ |e_2|) \\ [\lambda\alpha{:}|\star|.|\sigma|]\ |\ e_{\mathit{int}}\ |\ |\ e_{\rightarrow}\ | \end{array}$$

which steps to

$$\begin{array}{l} (\Lambda\gamma{:}|\star| \rightarrow \star.\lambda i{:}(\gamma|\mathit{int}|).\lambda a{:}(\forall\alpha.\gamma\alpha \rightarrow \forall\beta.\gamma\beta \rightarrow \gamma|\alpha \rightarrow \beta|). \\ \quad a\ [|\tau_1|]\ (|e_1|\ [\gamma]\ i\ a)\ [|\tau_2|]\ (|e_2|\ [\gamma]\ i\ a)) \\ [\lambda\alpha{:}|\star|.|\sigma|]\ |\ e_{\mathit{int}}\ |\ |\ e_{\rightarrow}\ | \end{array}$$

which steps to the translation of the right hand side.

$$\begin{array}{ll} |\ e_{\rightarrow}\ | & [|\tau_1|]\ (|e_1|\ [\lambda\alpha{:}|\star|.|\sigma|]\ |\ e_{\mathit{int}}\ |\ |\ e_{\rightarrow}\ |) \\ & [|\tau_2|]\ (|e_2|\ [\lambda\alpha{:}|\star|.|\sigma|]\ |\ e_{\mathit{int}}\ |\ |\ e_{\rightarrow}\ |) \end{array}$$

□

As well as being correct, it is important that our translation be interesting. What does this property mean? The target language of the translation could have been the trivial language with only one element. Even though the obvious translation to that language is trivially correct (in the sense defined above) the target language does not capture any of the properties of the source language.

The property that we care about the source language is that it allows us to write type-analyzing operations, while enforcing that those operations use type information correctly. Another possible translation might send the type representations to a datatype like structure in Scheme and "forget" the type-dependency of the R type. For example, we might represent R_{int} with a scheme symbol:

```
(define rint 'Rint)
(define rint? (lambda (rep) (eq? rep 'Rint)))
```

We might also represent $R_{\rightarrow}$ with a scheme list, tagged with a symbol at the head. For ease of use, we can also define functions to access the components of that list, once we know the term is a representation of the arrow type.

```
(define rarrow (lambda (e1 e2) (list 'Rarrow e1 e2)))
(define rarrow? (lambda (rep) (and (list? rep)
                                   (not (null? rep))
                                   (eq? (car rep) 'Rarrow))))
(define get-e1 cadr)
(define get-e2 caddr)
```

With these definitions, we can implement a Scheme typerec term.

```
(define typerec
   (lambda (rep eint earrow)
      (cond [(rint? rep)]
            [(rarrow? rep)
                 (earrow (typerec (get-e1 rep) eint earrow)
                         (typerec (get-e2 rep) eint earrow))])))
```

This translation has the same correctness properties as the one presented in this chapter. The image of all LIR code will type check in Scheme (all Scheme code type checks) and it will operate in the same way. However, when using these scheme definitions, there is nothing in Scheme that guarantees that we will use them correctly. The problem is that code that *does not* type check in LIR, will still type check in the Scheme version. So these definitions are not appropriate for writing our type-analyzing operations.

However, for the translation of LIR into LU, it is the case that LIR type errors will translate to LU type errors. I show this result by proving the contrapositive. Any translation of an LIR term that type checks in LU will also type check in LIR.

Before proving this result, I first assert a property about the translation:

Lemma 5.3.3 (Injectivity) *1. If $|\kappa_1| = |\kappa_2|$, then $\kappa_1 = \kappa_2$.*

2. If $|\sigma_1| = |\sigma_2|$ then $\sigma_1 = \sigma_2$.

3. (Corollary of 1) If $|\Delta_1| = |\Delta_2|$ then $\Delta_1 = \Delta_2$.

4. (Corollary of 2) If $|\Gamma_1| = |\Gamma_2|$ then $\Gamma_1 = \Gamma_2$.

Theorem 5.3.4 (Static adequacy) *1. If $\Delta \vdash_U |c| : \kappa$ then there exists Δ_R and κ_R such that $|\Delta_R| = \Gamma$ and $|\kappa_R| = \kappa$ and $\Gamma_R \vdash_R c : \kappa_R$.*

2. If $\Delta \vdash_U |\sigma| : \kappa$ then there exists Δ_R such that $|\Delta| = \Delta_R$ and $\Delta_R \vdash_R \sigma$.

3. If $\Delta; \Gamma \vdash_U |e| : \tau$ then there exists Δ_R, Γ_R and σ_R such that $|\Delta_R| = \Delta$, $|\Gamma_R| = \Gamma$ and $|\sigma_R| = \tau$ and $\Delta_R; \Gamma_R \vdash_R e : \sigma_R$.

Proof

Proof is by structural induction of e. I show the case for *typerec* in the proof of part 3 below. Suppose e is $\mathit{typerec}^{it}[\alpha.\sigma]\ e_1\ \ e_{int}\ \ e_{\rightarrow}$ and that

$$\Delta; \Gamma \vdash_U |e_1|\ [\lambda\alpha{:}|\star|.|\sigma|]\ |\ e_{int}\ |\ |\ e_{\rightarrow}\ | : \tau$$

Below, let $\gamma = \lambda\alpha{:}|\star|.|\sigma|$. By inversion

$$\begin{array}{ll}
\Delta; \Gamma \vdash_U |e_1| : \forall\beta{:}|\star| \rightarrow \kappa.\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 & \text{where } \tau = \tau_3[\gamma/\beta] \\
\Delta \vdash_U \lambda\alpha{:}|\star|.|\sigma| : |\star| \rightarrow \kappa & \\
\Delta, \alpha{:}|\star| \vdash_U |\sigma| : \kappa & \\
\Delta; \Gamma \vdash_U |\ e_{int}\ | : \tau_1[\gamma/\beta] & \\
\Delta; \Gamma \vdash_U |\ e_{\rightarrow}\ | : \tau_2[\gamma/\beta] &
\end{array}$$

By induction,

$$\begin{array}{ll}
\Delta_R, \alpha{:}\star \vdash_R \sigma & \text{where} \\
\Delta_R; \Gamma_R \vdash_R e_1 : \sigma_0 & |\sigma_0| = \forall\beta{:}|\star| \rightarrow \kappa.\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \\
\Delta_R; \Gamma_R \vdash_R e_{int} : \sigma_1 & |\sigma_1| = \tau_1[\gamma/\beta] \\
\Delta_R; \Gamma_R \vdash_R e_{\rightarrow} : \sigma_2 & |\sigma_2| = \tau_2[\gamma/\beta]
\end{array}$$

Note that by injectivity all of the above contexts are the same. As $\Delta_R, \alpha{:}\star \vdash_R \sigma$, then κ must be $\star$.

Now consider the possible identities of σ_0. Since we must produce a polymorphic type in the translation, σ_0 must either itself be a polymorphic type, or must be $R\tau'$ for some τ'. However, if σ_0 were a polymorphic type, what is the kind of the type variable? There is no LIR kind that translates to $|\star| \to \star$. So σ_0 must be $R\tau'$.

$$
\begin{aligned}
|\sigma_0| &= \forall\gamma{:}|\star| \to \star.\gamma|\,\mathit{int}\,| \to (\forall\delta{:}|\star|.\gamma\delta \to \forall\eta{:}\star.|\eta| \to \gamma|\delta \to \eta|) \to \gamma|\tau'| \\
&= \forall\beta{:}|\star| \to \kappa.\tau_1 \to \tau_2 \to \tau_3
\end{aligned}
$$

so

$$
\begin{aligned}
|\sigma_1| = \tau_1[\gamma/\beta] &= \gamma|\,\mathit{int}\,| \\
&= |\sigma[\mathit{int}/\alpha]| \\
|\sigma_2| = \tau_2[\gamma/\beta] &= (\forall\delta{:}|\star|.\gamma\delta \to \forall\eta{:}\star.\gamma\eta \to \gamma|\delta \to \eta|) \\
&= |\forall\delta{:}\star.\sigma[\delta/\alpha] \to \forall\eta{:}\star.\sigma[\eta/\alpha] \to \sigma[\delta \to \eta/\alpha]| \\
\tau_3[\gamma/\beta] &= \gamma|\tau'| \\
&= |\sigma[\tau'/\alpha]|
\end{aligned}
$$

$\Delta_R, \alpha{:}\star \vdash_R \sigma$
$\Delta_R; \Gamma_R \vdash_R e_1 : R\tau'$
$\Delta_R; \Gamma_R \vdash_R e_{\mathit{int}} : \sigma[\mathit{int}/\alpha]$
$\Delta_R; \Gamma_R \vdash_R e_{\to} : \forall\delta{:}\star.\sigma[\delta/\alpha] \to \forall\eta{:}\star.\sigma[\eta/\alpha] \to \sigma[\delta \to \eta/\alpha]$
then we may conclude
$\Delta_R; \Gamma_R \vdash_R \mathit{typerec}^{it}[\alpha.\sigma]\ e_1\ e_{\mathit{int}}\ e_{\to} : \sigma[\tau'/\alpha]$

□

5.4 Discussion

Despite the simplicity of this encoding, it falls short for a number of reasons. First, it probably is not possible to extend this encoding to an R-constructor or extend *typerec* to the primitive recursion of the full LIR language. Second, the target language has different properties than the source language.

5.4.1 Extension to an R-constructor

It is also possible to formulate the LIR language so that as well as an R-type, it also has an R-constructor [CWM98]. In this case, the version of LIR needs to have a constant that represents this constructor R_R, and a branch in *typerec* that matches this constant. With the R-constructor, type analysis is more complete.

There are fewer elements of the type language that may not be represented by constructors, and therefore not analyzed by *typerec*.

However, we cannot encode the R-type if we support R-constructors. Adding an additional constructor is no problem—it is the branch for the R-constructor in the encoding of the R-type is the source of the difficulty. The type of this branch in *typerec*$[\alpha.\sigma]$ should be

$$[\alpha.\sigma]\langle R : \star \to \star\rangle = \forall\beta{:}\star.\sigma[\beta/\alpha] \to \sigma[R\beta/\alpha]$$

By giving the branch this type, we must change the translation of the R-type so that includes type of this branch. In that case, $|R\tau|$ must be a recursive definition:

$$\begin{aligned}&\forall\gamma{:}|\star| \to \star.\ \gamma|\,int\,|\\ &\quad \to (\forall\alpha{:}|\star|.\gamma\alpha \to \forall\beta{:}|\star|. \to \gamma\beta \to \gamma|\alpha \to \beta|)\\ &\quad \to (\forall\beta{:}|\star|.\gamma\alpha \to \gamma|R\beta|)\\ &\quad \to \gamma|\tau|\end{aligned}$$

We have defined $|R\tau|$ in terms of $|R\beta|$. We need a recursive type to represent this definition. As LU does not include them, we cannot extend the encoding. However, this restriction is not that limiting as we might expect that a realistic term language include recursive types.

5.4.2 Extension to primitive recursion

At the term level we could extend the previous definition of $typerec^{it}$ to a primitive recursive version $typerec^{pr}$ by providing terms of type $R\alpha$ and $R\beta$ to $e_{\to}$. In that case, again $|R\tau|$ must be a recursive definition:

$$\begin{aligned}&\forall\gamma{:}|\star| \to \star.\ \gamma|\,int\,|\\ &\quad \to (\forall\alpha{:}|\star|.R\alpha \to \gamma\alpha \to \forall\beta{:}|\star|.R\beta \to \gamma\beta \to \gamma|\alpha \to \beta|)\\ &\quad \to \gamma|\tau|\end{aligned}$$

With the addition of parameterized recursive types, the definition of $typerec^{pr}$ is no more difficult than that of $typerec^{it}$; just supply the extra arguments to the arrow branch. In other words,

$$\begin{aligned}|\,R_{\to}\,| = \ &\Lambda\alpha{:}|\star|.\Lambda\beta{:}|\star|.\lambda x_1{:}|R\alpha|.\lambda x_2{:}|R\beta|.\\ &\quad\Lambda\gamma{:}|\star \to \star|.\lambda i.\lambda a.\\ &\quad\quad a[\alpha][\beta]\ x_1\ x_2\ (x_1[\gamma]ia)(x_2[\gamma]ia)\end{aligned}$$

However, we cannot add recursive kinds to implement primitive recursion at the type constructor level without losing decidable type checking. Even without resorting to recursion, there is another well-known technique for encoding primitive recursion in terms of iteration: pairing the argument with the result during the iteration.[3] Unfortunately, this pairing trick only works for closed expressions, and only produces terms that are $\beta\eta-$equivalent in the target language. Therefore, at the term level, our strong notion of dynamic correctness does not hold. Using this technique, we must weaken it to:

If $\emptyset \vdash_U e : \sigma$ and $e \mapsto e'$ then $|e|$ is $\beta\eta$-convertible with $|e'|$.

At the type-constructor level, $\beta\eta$-equivalence is sufficient. However, for type checking, we need the equivalence to extend to constructors with free-variables. The reason that this trick does not work is that LU can encode iteration over datatypes only weakly; there is no induction principle for this encoding provable in LU. Therefore, we cannot derive a proof of equality in the equational theory of the target language that relies on induction. This weakness has been encountered before. In fact, it is conjectured that it is impossible to encode primitive recursion in System F using $\beta\eta$-equality [SU99]. A stronger equational theory for LU, perhaps one incorporating a parametricity principle [PA93], might solve this problem. However, a simpler way to support primitive recursion would be to include an operator for primitive recursion directly in the language as we did in LX [Men87, CPM88, PPM90, CW99a].

5.4.3 Impredicativity and non-termination

Another issue with this encoding is that the target language must have impredicative polymorphism at the type and kind level. In practice, this property is acceptable in the target language. Although impredicativity at the kind level destroys strong-normalization of the term level [Coq94],[4], intensional polymorphism was designed for typed compilation of Turing-complete language [HM95]. Furthermore, Trifonov et al. show that impredicative kind polymorphism aids in the analysis of quantified types [TSS00][5]. Also, impredicativity at the type level is vital for such transformations as typed closure conversion. Allowing such impredicativity in the source language does not prevent this encoding; we can similarly encode the type-erasure version of their language [STS00].

However, the source language of this paper, LIR, is predicative and strongly normalizing, and the fact that this encoding destroys these properties is unsatis-

[3]See the tutorials in Meertens [Mee92] and Mitchell [Mit96] Section 9.3

[4]Coquand [Coq94] originally derived a looping term by formalizing a paradox along the lines of Reynolds' theorem [Rey84], forming an isomorphism between a set and its double power set. Hurkens [Hur95] simplified this argument and developed a shorter looping term, using a related paradox.

[5]See 6.6.4 for more discussion on this point.

factory. It seems reasonable, then, to look at methods of encoding iteration within predicative languages [PM93, Dyb91]. In adding iteration to the kind level, *strict* positivity (the recursively bound variable may not appear to the left of an arrow) may be required [CPM88], to prevent the definition of an equivalent paradox.

5.4.4 Related work

The language Haskell [PH99] uses an alternative way to implement type-analyzing functions (or ad-hoc polymorphism). In Haskell, type classes [WB89] declare what operations are available to abstract types. At run-time, instead of passing the representation of a type, instead polymorphic functions pass dictionaries including implementations of the operations at that type. In some sense, an R-type is a "universal" dictionary allowing the definition of any operation from any class.

In Standard ML [MTHM97], Yang [Yan98] similarly used it to encode type-specialized functions (such as type-directed partial evaluation [Dan96]). Because core ML does not support higher-order polymorphism, he presented his encoding within the ML module system.

Chapter 6

Higher-order type analysis

In the last two chapters, I argued that type analysis was merely a programming idiom. With an expressive type constructor language, a type-analyzing operator may be encoded. In this chapter, I turn the attention of type analysis to the full constructor language—instead of interpreting just the types, what if typerec could also interpret the type constructors in a principled way?

6.1 Polytypic programming

The idea of polytypic programming is to define functionality using type structure instead of values. As I discussed in Chapter 1, classic examples of polytypic programming include pretty printers, debuggers, equality functions and mapping functions. For example, by examining the type of a term, a polytypic pretty-printer can break the term into basic parts, and can print arbitrarily complex data-structures using this decomposition. The theory behind describing such polytypic operations has been explored in a variety of frameworks [WB89, ACPP91, She93, ACPR95, DRW95, HM95, JJ97, JBM98, Rue98, CW99a, Hin00, TSS00].

Nevertheless, no single existing framework encompasses all polytypic programs. These systems are limited by what polytypic operations they may express and by what types they may examine. These deficiencies are unfortunate because advanced languages depend crucially on these features. Only some frameworks for polytypism may express operations over parameterized data structures, such as maps and folds [JJ97, JBM98, Rue98, Hin00]. Yet parametric polymorphism is essential to modern typed programming languages. It is intrinsic to functional programming languages, such as ML [MTHM97] and Haskell [PH99], and also extremely important to imperative languages such as Ada [ISO94] and Java [BOSW98, GJS96]. Furthermore, only some frameworks for polytypism may examine types with binding structure, such as polymorphic or existential types [ACPR95, CW99a, TSS00]. The LX language of Chapter 4 has this capability. However, these types are becoming increasingly more important. Current implementations of the Haskell language [JR99, GHC02] include a form of existential type and first class polymorphism. Existential types are particularly useful for implementing dynamically extensible systems that may be augmented at run time with new operations and new types of data [HWC01]. Also, the extension of polytypic programming to an object-oriented language will require the ability to examine types with binding structure.

What is necessary to accommodate all types and all operations? First, because a quantified type is not known until run time, a type-passing (like LI) or representation-passing (like LIR) interpretation is required to examine types with binding structure. By incorporating the type information with the execution of the language, such frameworks may define polytypic operations over abstract types. Second, the class of polytypic operations including mapping functions, reductions, zipping functions and folds must be defined in terms of higher-order type constructors instead of types. Such type constructors are "functions" such as *list* or *tree*, that are parameterized by other types.

There is no reason why one system should not be able to define polytypic operations over both higher-order type constructors and quantified types. In fact, the two abilities are quite complementary when quantified types are represented using higher-order type constructors (i.e., with higher-order abstract syntax [Chu40, PE88, TSS00]). For example, the constant $\forall_\star$ applied to the type function $(\lambda\alpha{:}\star.\alpha \to \alpha)$ represents the polymorphic type $\forall\alpha.\alpha \to \alpha$.

In this chapter, I address the previous limitations of polytypic programming and demonstrate how well these abilities fit together by extending LI (from Chapter 2) to higher-order polytypism. Recall, in LI polytypic operations are defined by run-time examination of the structure of first-order types with the special term *typerec*. An analyzable type is either *int*, *string*, a product type composed of two other types, or a function type composed of two other types. As these simple type constructors form an inductive datatype, *typerec* defines a fold (or catamorphism) over its type argument. For example, the result of analyzing types such as $\tau_1 \times \tau_2$ is defined in terms of analyses of τ_1 and τ_2.

With the inclusion of type constructors that take a higher-order argument (such as $\forall_\star$ with argument of kind $\star \to \star$) the type structure of the language is no longer inductive. Previously, Trifonov et al. [TSS00] avoided this issue by using the kind-polymorphic type constructor $\forall$ of kind $\forall\chi.(\chi \to \star) \to \star$ instead of $\forall_\star$ to represent polymorphic types. As the argument of $\forall$ does not have a negative occurrence of the kind $\star$, the type structure remains inductive.

Hinze [Hin00] defined polytypic operations over type constructors by viewing a polytypic definition as an *interpretation* of the entire type constructor language, instead of a fold over the portion of kind type. However, his framework is based on compile-time definitions of polytypic functions (as opposed to run-time type analysis) and so cannot instantiate these functions with polymorphic or existential types. Here, I use this idea to extend Harper and Morrisett's *typerec* to a run-time interpreter for the type language, so that it may analyze higher-order type constructors and quantified types.

In the rest of this section, I review LI and Hinze's framework for polytypic programming. In Section 6.2 I extend *typerec* to constructors of function kind. Because a polytypic definition is a model of the type language, it inhabits a unary *logical relation* indexed by the kind of the argument type constructor. A simple generalization in Section 6.3 extends this *typerec* to inhabit multiplace logical relations. Furthermore, in Section 6.4 I generalize *typerec* to constructors of polymorphic kind. This extension admits the analysis of the $\forall$ constructor and encompasses as a special case the previous approach of Trifonov et al. [TSS00]. Also, incorporating kind polymorphism enables further code sharing; without it, polytypic definitions must be duplicated for each kind of type argument. Finally, in Section 6.5 I compare this approach with other systems.

6.1.1 Higher-order polytypism

Why do I need to extend *typerec* to higher-order constructors? With the semantics of LI, it may not express all polytypic definitions. For example, I cannot use it to define the function *fsize* that counts up the number of values of type β in a data structure of type $T(\alpha\ \beta)$. This functions should be of type $\forall\alpha{:}\star \to \star.\forall\beta{:}\star\,.\,T(\alpha\ \beta) \to \mathit{int}$. For example, if $c_1 = \lambda\alpha{:}\star\,.\alpha \times \mathit{int}$ and $c_2 = \lambda\alpha{:}\star\,.\alpha \times \alpha$, then $\mathit{fsize}[c_1]$ and $\mathit{fsize}[c_2]$ are constant functions returning 1 and 2 respectively. If α is instantiated with *list*, $\mathit{fsize}[\mathit{list}]$ is the standard length function.

Recall the typing rule for LI *typerec*, below:

$$[\textit{e-trec}]\quad \frac{\begin{array}{c}\Delta \vdash c : \star \\ \Delta, \alpha{:}\star \vdash \sigma \\ \Delta; \Gamma \vdash e_\oplus : [\alpha.\sigma]\langle \oplus : \kappa_\oplus \rangle \qquad \forall \eta_\oplus \in \overline{e}\end{array}}{\Delta; \Gamma \vdash \mathit{typerec}[\alpha.\sigma]\, c\, \overline{e} : \sigma[c/\alpha]}$$

Again, the symbol $\overline{e}$ abbreviates the branches of the *typerec* ($\mathit{int} \Rightarrow e_{\mathit{int}}, \to\, \Rightarrow e_{\to}, \times \Rightarrow e_{\times}$). In this chapter I will be deliberately vague about what type constructors comprise these branches and add new branches as necessary. I use $\oplus$ to notate arbitrary type constructor constants (such as $\mathit{int}, \to, \times$, called operators), and assume each $\oplus$ is of kind $\kappa_\oplus$. To verify the branches of the *typerec*, this rule depends on the following definition of a *polykinded type*, written $[\alpha.\sigma]\langle\!\langle c : \kappa\rangle\!\rangle$. This type represents the result of a branch on constructor c of kind κ. It is defined below by induction on κ

$$[\alpha.\sigma]\langle\!\langle c : \star\rangle\!\rangle = \sigma[c/\alpha]$$
$$[\alpha.\sigma]\langle\!\langle c : \kappa_1 \to \kappa_2\rangle\!\rangle = \forall\alpha{:}\kappa_1.[\alpha.\sigma]\langle\!\langle \alpha : \kappa_1\rangle\!\rangle \to [\alpha.\sigma]\langle\!\langle (c\alpha) : \kappa_2\rangle\!\rangle$$

In the above rule, the argument to *typerec* must be of kind $\star$. If α is an unknown type constructor of kind $\star\to\star$, it cannot directly be the argument to *typerec*. It is possible to analyze the result of applying it, but that may not work. For example, an attempt to define *fsize* might start out as

$$\mathit{fsize} = \Lambda\alpha{:}\star \to \star.\Lambda\beta{:}\star\,.\,\mathit{typerec}[\lambda\gamma.\gamma \to \mathit{int}](\alpha\ \beta)\ \mathit{of}\ \ldots$$

However, this approach is wrong. At run time, β will be instantiated before *typerec* analyzes $(\alpha\ \beta)$. The value returned by *typerec* will depend on what type instantiated β. If that type was *int*, then $c_1\ \mathit{int}$ and $c_2\ \mathit{int}$ will reduce to the same type, and analysis will produce the same result, even though c_1 and c_2 are different constructors. Therefore, in order to define *fsize*, we must somehow analyze the type constructor α independently of its argument β.

How should *typerec* analyze higher-order type constructors? What should the return type of such an analysis be? I draw inspiration from a recently proposed framework for generic programming by Ralf Hinze [Hin00]. Unlike in LI, where types are analyzed at run time, in this framework, polytypic functions are created and specialized to their type arguments at compile-time. The key insight is that each polytypic definition should be an *interpretation* of the type language with elements of the term language. If this interpretation is sound—i.e. when two types are equal, their interpretations are equal—we will be able to reason about the behavior of a polytypic definition. Otherwise, the process of substituting equal types for equal types could affect the meanings of the programs. A sound interpretation of higher-order types is to interpret type functions as term functions and type application as term application, because the β-equality between types (i.e.$(\lambda\alpha{:}\kappa.c_1)c_2 = c_2[c_1/\alpha]$) will be preserved by β-equivalence in the term language. However, the constants of the type language $(\mathit{int}, \to, \times)$ may be mapped to any term (of the right type) providing the flexibility to define a number of different polytypic operations.

For example, the definition of the polytypic operation *size* is in Figure 6.1. This operation is defined by induction over a type constructors c. It is also parameterized by a finite map η (an environment) mapping type variables to terms. I use $\emptyset$ as the empty map, extend a map with a new mapping from the type variable α

$$
\begin{array}{lcl}
size\ \langle \alpha \rangle\ \eta & = & \eta(\alpha) \\
size\ \langle \lambda\alpha{:}\kappa.c \rangle\ \eta & = & \Lambda\alpha{:}\kappa.\ \lambda x{:}(Size\ \langle \kappa \rangle\ \alpha). \\
 & & \quad size\ \langle c \rangle\ (\eta, \alpha{:}x) \\
size\ \langle c_1 c_2 \rangle\ \eta & = & (size\ \langle c_1 \rangle\ \eta)\ [c_2]\ (size\ \langle c_2 \rangle\ \eta) \\
size\ \langle int \rangle\ \eta & = & \lambda x{:}\ int\ .0 \\
size\ \langle string \rangle\ \eta & = & \lambda x{:}\ string\ .0 \\
size\ \langle \times \rangle\ \eta & = & \Lambda\alpha{:}\star.\lambda x{:}(\alpha \to int).\ \Lambda\beta{:}\star.\lambda y{:}(\beta \to int). \\
 & & \quad \lambda v{:}(\alpha \times \beta).x(\pi_1 v) + y(\pi_2 v) \\
size\ \langle + \rangle\ \eta & = & \Lambda\alpha{:}\star.\lambda x{:}(\alpha \to int).\ \Lambda\beta{:}\star.\lambda y{:}(\beta \to int). \\
 & & \quad \lambda v{:}(\alpha + \beta).\ case\ v\ of\ (inj_1\ w \Rightarrow xw \mid inj_2\ w \Rightarrow yw)
\end{array}
$$

where

$$
\begin{array}{lcl}
Size\ \langle \star \rangle\ c & = & c \to int \\
Size\ \langle \kappa_1 \to \kappa_2 \rangle\ c & = & \forall\alpha{:}\kappa_1.Size\ \langle \kappa_1 \rangle\ \alpha \to Size\ \langle \kappa_2 \rangle\ (c\ \alpha)
\end{array}
$$

Figure 6.1: Example: *size* in Hinze's system

to the term e with the notation $\eta, \alpha{:}e$, and retrieve the mapping for a type variable with $\eta(\alpha)$. All variables in the argument of *size* should be in the domain of η. The first three lines of the definition in this table are common to polytypic definitions. The definition for variables is determined by retrieving the mapping of the variable from environment. The environment is extended in the definition of *size* for type functions ($\lambda\alpha{:}\kappa.c$). As a type function is of higher kind, it is defined to be a polymorphic function from the *size* of the type argument, to the *size* of the body of the type constructor, with the environment updated to provide a mapping for the type variable occurring in the body. The type of x is determined by the kind of α and is explained in the following. Because a type function maps to a polymorphic term function, a type application produces a term application.

The last four cases determine the behavior of *size*. Intuitively, *size* produces an iterator over a data structure, which adds the "sizes" of all of its parts. I would like to use this operation in the definition of *fsize* as follows. Because *list* is a type constructor, the specialization $size\langle list \rangle$ maps a function to compute the "size" of values of some type β, to a function to compute the "size" of the entire list of type $list\ \beta$. If we supply the constant function $\lambda x{:}\beta.1$ for the list elements, we produce the desired length function for lists. Therefore, I may define *fsize* specialized by any closed type constructor c as $\Lambda\beta{:}\star.(size\langle c\rangle\emptyset)\ [\beta]\ (\lambda x{:}\beta.1)$.[1] For base types, such as *int* or *string*, $size$ produces the constant function $\lambda x.0$, because they should not be included in computing the size. The type constructors $+$ and $\times$ are both parameterized by the two subcomponents of the $+$ or $\times$ types (α and β) and functions to compute their sizes (x and y).

For example, the slightly simplified specialization of $size\langle\lambda\alpha.\alpha\times string\rangle\emptyset$, when all of the definitions have been applied, is below. It is a function that when given an argument to compute the size of terms of type α, should accept a pair and apply this argument to the first component of the pair. (As the second component of the pair is of type *string*, its $size$ is 0).

$$
\begin{array}{l}
size\langle\lambda\alpha.\alpha\times string\rangle\emptyset \\
= \quad \{\text{ using the definition for type abstraction }\} \\
\Lambda\alpha{:}\star.\lambda w{:}\alpha\rightarrow int\,.size\langle\alpha\times string\rangle,\alpha{:}w \\
= \quad \{\text{ definition for application, applied twice }\} \\
\Lambda\alpha{:}\star.\lambda w{:}\alpha\rightarrow int\,.(size\langle\times\rangle,\alpha{:}w) \\
\quad [\alpha]\ (size\langle\alpha\rangle,\alpha{:}w)\ [string]\ (size\langle string\rangle,\alpha{:}w) \\
= \quad \{\text{ definitions for }\times\text{, variables, and } string\ \} \\
\Lambda\alpha{:}\star.\lambda w{:}\alpha\rightarrow int\,.(\Lambda\alpha{:}\star.\lambda x{:}\alpha\rightarrow int\,.\Lambda\beta:\star.\lambda y{:}\beta\rightarrow int\,. \\
\quad \lambda v:\alpha\times\beta.x(\pi_1 v)+y(\pi_2 v))[\alpha]\ w\ [string]\ (\lambda x{:}\,string\,.0) \\
= \quad \{\ \beta\text{-simplification }\} \\
\Lambda\alpha{:}\star.\lambda w{:}(\alpha\rightarrow int).\lambda v{:}(\alpha\times string).w(\pi_1 v)+0
\end{array}
$$

Because type functions are mapped to term functions, the *type* of the polytypic definition (such as $size$) will be determined by the *kind* of the type constructor analyzed. In each instance, the definition of $size\langle c\rangle$ will be of type $Size\langle\kappa\rangle c$ where κ is the kind of c and $Size\langle\kappa\rangle c$ is defined by induction on the structure of κ. If the constructor c is of kind $\star$, then $Size\langle\star\rangle c$, is a function type from c to *int*. Otherwise, if c is of higher kind then $size$ is parameterized by a corresponding $size$ argument for the type argument to c.

$$
\begin{array}{l}
Size\langle\star\rangle c = c\rightarrow int \\
Size\langle\kappa_1\rightarrow\kappa_2\rangle c = \forall\alpha{:}\kappa_1.Size\langle\kappa_1\rangle\alpha\rightarrow Size\langle\kappa_2\rangle(c\alpha)
\end{array}
$$

Why does the definition of $size$ make sense? It is defined over the syntax of a type, but a type is actually an equivalence class of syntactic expressions. To be well defined, a polytypic function must return equivalent terms for all equivalent types, no matter how the types are expressed. For example, $size$ instantiated with $(\lambda\alpha{:}\star$

[1]Unlike LI where types are analyzed at run time, in this framework polytypic functions are created and specialized to their type arguments at compile-time, so I may not make *fsize*$\langle c\rangle$ polymorphic over c.

$.\alpha \times string)\, int$ must be equal to $size\ \langle int \times string \rangle$ because these two types are equal by β-equality. Because the term functions provide the necessary equational properties, the definition of *size* is sound. Therefore, though the interpretations of the type operators $(int, \rightarrow, \times)$ may change for each polytypic operation, the interpretations of functions $(\lambda\alpha{:}\kappa.c)$, variables α, and applications $(c_1 c_2)$ remain constant in every polytypic definition. As a result, the types of polytypic operations can be expressed using the same notation for polykinded types that I used to describe the type of each branch of *typerec*. For example, I express $Size\langle\kappa\rangle c$ in this notation as $[\alpha.\alpha \rightarrow int]\langle\!| c : \kappa \rangle$.

6.2 The semantics of higher-order *typerec*: The LH language

Hinze's framework specifies how to define a polytypic function at compile time by translating closed types into terms. However, in some cases, such as in the presence of polymorphic recursion, first-class polymorphism, or separate compilation it may not be possible to specialize all type abstractions at compile time. Therefore, in this section, I extend a language supporting run-time type analysis to polytypic definitions over higher-order type constructors. I do so by changing the behavior of LI's *typerec* to an *interpret* the type language at run time.

Just as each of the branches in the definition of *size* are described by polykinded types, so are each of the branches of *typerec*. Carrying the analogy further suggests that I may extend *typerec* to all type constructors by relaxing the restriction that the argument to *typerec* be of kind $\star$, and by using a polykinded type to describe the result of *typerec*.

$$\frac{\begin{array}{c}\Delta \vdash c : \boxed{\kappa} \\ \Delta, \alpha{:}\star \vdash \sigma \\ \Delta ; \Gamma \vdash e_\oplus : [\alpha.\sigma]\langle\!| \oplus : \kappa_\oplus \rangle \qquad (\forall e_\oplus \in \overline{e})\end{array}}{\Delta ; \Gamma \vdash typerec[\alpha.\sigma]\, c\ \ of\ \ \overline{e} : \boxed{[\alpha.\sigma]\langle\!| c : \kappa \rangle}}$$

Unfortunately, this judgment is not complete. As in the definition of $size\langle c \rangle \eta$, the operational semantics for higher-order *typerec* must involve some sort of environment η, and the typing judgment must describe that environment.

In the following, I introduce higher-order *typerec*, first presenting its operational semantics, and then describe how to type-check a *typerec* term. I conclude this section by exhibiting the expressiveness of a language including this term with a number of examples demonstrating *typerec* extended to type constructors with binding constructs.

Table 6.1: LH: Syntax

$(kinds)$	κ	$::=$	$\star \mid \kappa_1 \rightarrow \kappa_2$
(ops)	$\oplus$	$::=$	$int \mid \rightarrow \mid \times \mid + \mid \ldots$
$(con's)$	c	$::=$	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \oplus$
$(types)$	σ	$::=$	$T(c) \mid int \mid \sigma \rightarrow \sigma \mid \forall\alpha{:}\kappa.\sigma \mid \ldots$
$(exps)$	e	$::=$	$i \mid x \mid \lambda x{:}\sigma.e \mid e_1 e_2 \mid \mathit{fix}\ x{:}\sigma.e \mid \Lambda\alpha{:}\kappa.e \mid e[c]$
		$\mid$	$\mathit{typerec}[\alpha.\sigma][\Delta, \eta, \rho]\ c\ \ \mathit{of}\ \ \overline{e}\ \ \mid \ldots$

To make the examples concrete, I replace the *typerec* term in Harper and Morrisett's LI with higher-order *typerec*. The syntax of this language appears in Table 6.1; the semantics not involved with *typerec* remains the same. Type constructors and types are again separate syntactic classes in this language, with an injection $T(c)$ between the type constructors of kind $\star$, and the types.

I define the operational semantics for higher-order *typerec* by induction on the structure of the type constructor argument, c, at the bottom of Table 6.2. In order to interpret type variables in this argument, I add an environment component η to *typerec*. The intention is that the free type variables in c will be in the domain of η, and *typerec* will use it to look up the appropriate value when analysis reaches one of these variables. However, in order to define a sound operational semantics, I must be careful that these free type variables in c do not escape their scope. When part of c is used for a purpose other than type analysis (as in the rule for type application below) I must substitute away all of the free type variables occurring in c. For this substitution, I add to *typerec* an additional environment, ρ, mapping type variables to types. The notation $\rho(c)$ applied the substitution for all free variables of c in the domain of ρ.

When the argument to *typerec* is a type variable, the result is its mapping in the environment η. When type analysis reaches a type-constructor abstraction, both the term and the type environments are extended with the appropriate mappings. For a type application, *typerec* applies the analyzed constructor function to the analyzed argument. For operators, *typerec* just returns the appropriate branch.

A reassuring property of this *typerec* is that it derives the original operational rules. For example, the original version of *typerec* has the following evaluation rule for product types:

$$\begin{array}{l} \mathit{typerec}[\alpha.\sigma]\ (c_1 \times c_2)\ \ \mathit{of}\ \ \overline{e} \\ \quad \mapsto_i e_\times[c_1]\ (\mathit{typerec}[\alpha.\sigma]\ c_1\ \ \mathit{of}\ \ \overline{e})\ [c_2]\ (\mathit{typerec}[\alpha.\sigma]\ c_2\ \ \mathit{of}\ \ \overline{e}) \end{array}$$

With higher-order type analysis, because $c_1 \times c_2$ is the operator $\times$ applied to c_1 and c_2, the rule for type-constructor application generates the same behavior.

Table 6.2: LH: semantics for higher-order *typerec*

$\boxed{\Delta; \Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho}$

[*ctx-empty*]
$$\frac{}{\Delta; \Gamma[\alpha.\sigma] \vdash \emptyset \mid \emptyset \mid \emptyset}$$

[*ctx-add*]
$$\frac{\begin{array}{c}\Delta; \Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho \\ \Delta \vdash c : \kappa \\ \Delta; \Gamma \vdash e : [\alpha.\sigma]\langle c : \kappa \rangle\end{array}}{\Delta; \Gamma[\alpha.\sigma] \vdash \Delta', \beta{:}\kappa \mid \eta, \beta{:}e \mid \rho, \beta{:}c} \; (\beta \notin Dom(\Delta, \Delta'))$$

$\boxed{\Delta; \Gamma \vdash e : \sigma}$

[*e-trec*]
$$\frac{\begin{array}{c}\Delta, \alpha{:}\star \vdash \sigma \\ \Delta; \Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho \\ \Delta, \Delta' \vdash c : \kappa \\ \Delta; \Gamma \vdash e_\oplus : [\alpha.\sigma]\langle \oplus : \kappa_\oplus \rangle \qquad (\forall e_\oplus \in \overline{e})\end{array}}{\Delta; \Gamma \vdash \mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\, c \;\; \mathit{of} \;\; \overline{e} : [\alpha.\sigma]\langle \rho(c) : \kappa \rangle}$$

$\boxed{e \mapsto e'}$

[*ev-trec-var*]
$$\frac{}{\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \;\; \beta \;\; \mathit{of} \;\; \overline{e} \mapsto \eta(\beta)}$$

[*ev-trec-fn*]
$$\frac{}{\begin{array}{l}\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \;\; (\lambda\beta{:}\kappa.c) \;\; \mathit{of} \;\; \overline{e} \mapsto \\ \quad \Lambda\gamma{:}\kappa.\lambda x{:}[\alpha.\sigma]\langle \gamma : \kappa \rangle. \\ \qquad \mathit{typerec}[\alpha.\sigma][\Delta', \beta{:}\kappa, \eta, \beta{:}x, \rho, \beta{:}\gamma] \;\; c \;\; \mathit{of} \;\; \overline{e}\end{array}}$$

[*ev-trec-app*]
$$\frac{}{\begin{array}{l}\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \;\; (c_1 c_2) \;\; \mathit{of} \;\; \overline{e} \mapsto \\ \quad (\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \;\; c_1 \;\; \mathit{of} \;\; \overline{e}) \; [\rho(c_2)] \\ \qquad (\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \;\; c_2 \;\; \mathit{of} \;\; \overline{e})\end{array}}$$

Table 6.2 (Continued)

[*ev-trec-op*]	$\dfrac{}{\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \ \oplus \ \mathit{of} \ \ \overline{e} \mapsto e_{\oplus}}$

To typecheck a *typerec* term, I need a context Δ' to describe the kinds of the variables in the domain of η and ρ. I use this context as an additional assumption when checking the argument to *typerec*, and also employ it when checking η and ρ. For the latter, I formulate a new judgment $\Delta; \Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho$, stating that η and ρ are well formed. Intuitively this judgment states "in context $\Delta; \Gamma$, the environment η maps type variables in Δ' to appropriate terms for the result type annotation $[\alpha.\sigma]$, and the environment ρ maps those variables to type constructors of the same kind". This judgment is derived from two inference rules in Table 6.2. The first rule states that the empty context and the empty environments are always valid. In the second rule, if I add a new type variable α of kind κ to Δ', its mapping in ρ must be to a type constructor c also of kind κ, and its mapping in η must be to a term with type indexed by κ. Note that as I add to Δ' only type variables that are not in Δ, the domains of Δ and Δ' must be disjoint.

With this judgment, I can state the formation rule for higher-order *typerec* in Table 6.2, as an extension of the previous rule.

The LH version of *size* appears below. For each operator (*int*, *unit*,etc..) the branch in *typerec* is the same as in Hinze's definition in Figure 6.1. In this and following examples, when the maps annotating *typerec* are empty, they are elided.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{size} = \Lambda\alpha{:}\star \to \star.\ \mathit{typerec}[\beta.\beta \to \mathit{int}]\ \alpha\ \ \mathit{of}} \\
\quad \mathit{int} & \Rightarrow & \lambda y{:}\ \mathit{int}\ .0 \\
\quad \mathit{unit} & \Rightarrow & \lambda y{:}\ \mathit{unit}\ .0 \\
\quad \times & \Rightarrow & \Lambda\beta{:}\star.\lambda x{:}(\beta \to \mathit{int}).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to \mathit{int}). \\
 & & \lambda v{:}(\beta \times \gamma).\ x(\pi_1 v) + y(\pi_2 v) \\
\quad \to & \Rightarrow & \Lambda\beta{:}\star.\Lambda\gamma{:}\star.\mathit{undefined} \\
\quad + & \Rightarrow & \Lambda\beta{:}\star.\lambda x{:}(\beta \to \mathit{int}).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to \mathit{int}). \\
 & & \lambda v{:}(\beta + \gamma).\ \ \mathit{case}\ \ v\ \ \mathit{of}\ \ (\mathit{inj}_1\, z \Rightarrow x(z) \mid \mathit{inj}_2\, z \Rightarrow y(z))
\end{array}
$$

This example demonstrates a few deficiencies of the calculus presented so far. First, what about recursive types? The definition of *size* for lists and trees requires them. What about polymorphic or existential types? It seems reasonable to extend size to data types with abstract components. What about applying *size* to constructors of kind $\star \to \star$? This *typerec* can operate over constructors of any kind. I address these limitations in the rest of this chapter.

6.2.1 Recursive types

Higher-order type analysis is amenable to both *equi-recursive* and *iso-recursive* types. For simplicity, I begin with the non-parameterized version of these recursive types, created with the constructor $\mu_\star$ of kind $(\star \to \star) \to \star$.

Hinze's definitions were originally in the context of a language with *iso-recursive* types. In such a language, a recursive type is definitionally equal to its unrolling. The rules for type equivalence include the following rule to witness that fact.

$$\frac{\Delta \vdash c : \star \to \star}{\Delta \vdash \mu_\star c = c(\mu_\star c) : \star}$$

In order to preserve this equality, Hinze always translated polytypic functions over recursive types to recursive functions. Because a recursive function will unwind itself during execution, $size\langle \mu_\star c \rangle$ will step to $size\langle c \mu_\star \rangle$, and so both will provide the same behavior.

With this extra constraint on type equality, analysis of $(\mu_\star c)$ must be equal to that $c(\mu_\star c)$, just as analysis of $(\lambda\alpha{:}\kappa.c_1)c_2$ is equal to analysis of $c_1[c_2/\alpha]$. As in Hinze's definition, an evaluation rule for *typerec* in this context takes the fixed point of its argument as the interpretation of a recursive type[2].

$$\begin{array}{l} \mathit{typerec}[\alpha.\sigma][\Gamma, \eta, \rho]\ \mu_\star\ \overline{e}\ \mapsto \\ \quad \Lambda\beta{:}\star \to \star.\ \lambda x{:}[\alpha.\sigma]\langle \beta : \star \to \star \rangle. \\ \quad\quad \mathit{fix}\ f{:}[\alpha.\sigma]\langle \mu_\star\beta : \star \rangle.\ (x\ [\mu_\star\beta]\ f) \end{array}$$

However, type-checking in a systems with equi-recursive types is more complicated than in languages with iso-recursive types (such as LI, LX, etc..). Unlike those languages, we cannot use normalize-and-compare algorithm to decide when two type constructors are equivalent. Like the evaluation of recursive functions, unrolling a recursive type may not terminate. Efficient algorithms to decide equivalence (and subtyping) of regular recursive types do exists. Gapeyev, Levin and Pierce [GLP00] provide a tutorial, based on the work of Brandt and Henglein [BH97], Kozen et al.[KPS93], and Jim and Palsberg [JP99]. However, extending equivalence algorithms to parameterized (non-regular) recursive types is difficult. Determining when two of these types are equal is equivalent to the equivalence

[2]In a call-by-value calculus this rule is ill-typed because it takes the fixed point of an expression that is not necessarily of function type. To support this rule in such a calculus, the rule for *typerec* should require that σ return a function type for any argument.

problem for deterministic pushdown automata [Sol78]. That problem is known to be decidable [Sé97], but no tractable algorithm is known.

Iso-recursive types

In order to simplify the implementation of type equality, many languages support *iso-recursive* types: those that require explicit terms that coerce between a recursive type and its unfolding. In this framework, there is no equational rule for $\mu_\star$, but the calculus includes the two terms that witness the isomorphism between a recursive type and its unrolling. For non-parameterized recursive types, these terms have formation rules:

$$roll_{\mu_\star c} : c(\mu_\star c) \to \mu_\star c \qquad unroll : \mu_\star c \to c(\mu_\star c)$$

There is the most flexibility in the definition of polytypic functions with iso-recursive types. As there is no equivalence rule governing $\mu_\star$, polytypic functions are free to interpret it in any manner, as long as its branch in *typerec* has the correct type, determined by the kind of $\mu_\star$. This type is

$$[\alpha.\sigma]\langle\mu_\star : (\star \to \star) \to \star\rangle = \forall\beta{:}\star \to \star.(\forall\gamma{:}\star.\sigma[\gamma/\alpha] \to \sigma[\beta\gamma/\alpha]) \to \sigma[\mu_\star\beta/\alpha]$$

In most polytypic terms, the *typerec* branch for iso-recursive $\mu_\star$ will be the same as the evaluation rule for equi-recursive $\mu_\star$.[3] For example, the $\mu_\star$ branch for *size* is below. The difference between it and the rule for equi-recursive types is an η-expansion around $x[\mu_\star\alpha]f$ that allows us to insert the term coercion *unroll*.

$$\begin{array}{l} \mu_\star \Rightarrow \Lambda\alpha{:}\star \to \star. \\ \quad \lambda x{:}(\forall\beta{:}\star.T(\beta \to int) \to T(\alpha\beta \to int)). \\ \quad\quad fix\, f{:}T(\mu_\star\alpha \to int). \\ \quad\quad\quad \lambda y{:}T(\mu_\star\alpha).\ x\ [\mu_\star\alpha]\ f\ (unroll\, y) \end{array}$$

The argument α is the body of the recursive type, and the argument x is the result of *typerec* over that body. The definition of *size* for a recursive type should be a recursive function that accepts an argument, y, of a recursive type, unrolls it so that it is of type $T(\alpha(\mu_\star\alpha))$, and calls x to produce *size* for this object. The call to x needs an argument that computes the size of the parameter β to α—in this case, the parameter is $\mu_\star\alpha$, so the argument I need is the result I am computing in the first place. Therefore, I use *fix* to name this result f and supply it to x.

Now I have all of the pieces to write length for lists. The list type constructor is expressed with a recursive type as $\lambda\alpha{:}\star.\mu_\star(\lambda\beta{:}\star.unit + (\alpha \times \beta))$. As before,

[3] In Section 6.3.2 I discuss a term that is not.

the application $size[list]$ is of type

$$\forall\alpha{:}\star.T(\alpha \to int) \to T((list\ \alpha) \to int).$$

and to produce length, I supply the constant function one that will be employed at each node of the list.

$$length = \Lambda\alpha{:}\star.size[list][\alpha](\lambda x{:}\alpha.1)$$

Nested datatypes

While the types themselves grow more complicated, nothing much needs to change to generalize to parameterized recursive types. In this case, recursive types are created with the μ_κ type constructor below.

$$\mu_\kappa : ((\kappa \to \star) \to (\kappa \to \star)) \to (\kappa \to \star)$$

Parameterized recursive types are often useful in programming languages to describe *nested datatypes*. Nested datatypes are a powerful technique of representing constraints about the formation of datatypes [BM98, Oka98, BP99b, Oka99]. In general they may be described as parameterized datatypes in ML or Haskell in which the recursive type is applied to some type other than the parameterized variable. For example, even though Tree is a parameterized datatype when written in Haskell below,

```
data Tree a = Node | Leaf ((a, a), Tree a)
```

it is not a nested datatype as it may be described by a regular recursive type.

$$Tree = \lambda\alpha{:}\star.\mu_\star(\lambda\beta{:}\star.1 + ((\alpha \times \alpha) \times \beta))$$

A simple example of a nested datatype is a *power tree*. This datatype only represents complete binary trees. Power trees are defined in Haskell by the following datatype definition, where the recursive type `Pow` is applied to the pair `(a,a)` instead of just the type variable `a`.

```
data Pow a = Zero a | Succ (Pow (a,a))
```

We must use the $\mu_{\star\to\star}$ type constructor to describe power trees.

$$Pow = \mu_{\star\to\star}\ (\lambda\beta{:}\star \to \star.\lambda\alpha{:}\star.\alpha + \beta(\alpha \times \alpha))$$

Every power tree stores a complete binary tree. If the tree has 2^n elements, then the prefix of the data representation is the number n in unary.

```
p2 = Succ (Zero (1,2))
p4 = Succ (Succ (Zero ((1,2), (3, 4))))
p8 = Succ (Succ (Succ (Zero (((1,2),(3,4)), ((5,6),(7,8))))))
```

We can define a size function for power trees below, parameterized by `f` the size function for the type `a` as below.

```
powersize :: (a -> Int) -> (Pow a -> Int)
powersize f (Zero t) = f t
powersize f (Succ p) = powersize (\(x,y) -> f x + f y) p
```

If the power tree is the `Zero` constructor, we apply `f` to its argument, as it is of type `a`. Otherwise, if the tree is the `Succ` constructor, its argument is of type `Pow (a,a)`. Therefore we should call `powersize` recursively with a function from `(a,a)` to `Int`. Note that this function `powersize` requires polymorphic recursion in order to type check, as the recursive call to `powersize` is instantiated at the type `(a,a)` instead of just `a`).

Defining such operations over nested datatypes is a tricky process [BM98, BP99a]. However, by adding a branch for the type constructor $\mu_{\star\to\star}$ into the definition of *size*, then `powersize` may be developed automatically. If $\sigma = T(\alpha \to int)$, then $\mu_{\star\to\star}$ branch should be of type:

$$[\alpha.\sigma]\langle \mu_{\star\to\star} : ((\star \to \star) \to (\star \to \star)) \to \star \to \star\rangle$$

Expanding the definition, this type is:

$$\begin{array}{rl} & \forall\alpha{:}(\star \to \star) \to (\star \to \star).[\alpha.\sigma]\langle\alpha : (\star \to \star) \to \star \to \star\rangle \\ & \quad \to [\alpha.\sigma]\langle\mu_{\star\to\star}\alpha : \star \to \star\rangle \\ = & \forall\alpha{:}(\star \to \star) \to (\star \to \star).[\alpha.\sigma]\langle\alpha : (\star \to \star) \to \star \to \star\rangle \\ & \quad \to \forall\beta{:}\star\,.\,T(\beta \to int) \to T(\mu_{\star\to\star}\alpha\beta) \to int) \end{array}$$

The branch for *size* for this operator is very similar to that of μ, it uses *fix* to create a sizing function, g, for $\mu_{\star\to\star}$'s, then unrolls the argument, and uses r, instantiating β with $\mu_{\star\to\star}\alpha$ and then passing it g.

$$\begin{array}{l} \mu_{\star\to\star} \Rightarrow \Lambda\alpha{:}(\star \to \star) \to (\star \to \star). \\ \quad \lambda r{:}(\forall\beta{:}\star \to \star.[\alpha.\sigma]\langle\beta : \star \to \star\rangle \to \forall\gamma{:}\star\,.\,T(\gamma \to int) \to T(\alpha\beta\gamma) \to int). \\ \quad \mathit{fix}\ g : [\alpha.\sigma]\langle\mu_{\star\to\star}\alpha : \star \to \star\rangle. \\ \quad\quad \Lambda\gamma{:}\star\,.\ \lambda f{:}T(\gamma \to int).\ \lambda x{:}T(\mu_{\star\to\star}\alpha\gamma). \\ \quad\quad\quad r\ [\mu_{\star\to\star}\alpha]\ g\ [\gamma]\ f\ (\mathit{unroll}\,x) \end{array}$$

6.2.2 F2 polymorphism

The type constructor constants $\forall_\star$ and $\exists_\star$ (of kind $(\star \to \star) \to \star$) use higher-order abstract syntax [PE88] to describe polymorphic and existential types of F2 [Gir72, Rey83]. These types are a subset of the polymorphic and existential types of LI. They may only abstract types instead of constructors of any kind. The relationship between these type constructors and the corresponding types are:

$$\Delta \vdash T(\forall_\star c) = \forall \alpha{:}\star\,.\,T(c\alpha) \qquad \Delta \vdash T(\exists_\star c) = \exists \alpha{:}\star\,.\,T(c\alpha)$$

I can extend *size* with a branch for $\exists_\star$. In this branch, we need to provide a function to calculate the size of the hidden type, so I use the constant function zero:[4]

$$\begin{array}{l}\exists_\star \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda r{:}(\forall\beta{:}\star\,.\,T(\beta \to int) \to T(\alpha\beta \to int)).\\ \quad \lambda x{:}T(\exists\alpha).\,let\langle\beta, y\rangle = unpack\;x\;in\;\; r\;[\beta]\;\; (\lambda x{:}\beta.0)\;\; y\end{array}$$

With *size* I was fortunate that I could compute the value of *size* for the hidden type of an existential without analyzing it, as it was a constant function. However, for many polytypic functions, this is not the case, and the function I pass to operate on the hidden type is itself polytypic. In fact, often it is the polytypic function itself, called recursively. This is not surprising, considering the impredicative nature of $\forall_\star$ and $\exists_\star$ types: since the quantifiers range over *all* types I need an appropriate definition at all types.

For example, consider the simple function copy in Figure 6.2 that creates an identical version of its argument. At base types, it is an identity function, at higher types, it breaks apart its argument and calls itself recursively.

6.2.3 Typing properties of LH

The rules for the static and dynamic semantics are appropriate because they satisfy type preservation: looking at the four operational rules for *typerec*, we can see that no matter which one applies, if the original term was well-typed then the resulting term is also well-typed with the same type. Furthermore, a closed, well-typed *typerec* term is never stuck; for any type constructor argument, one of the four operational rules must apply. These two properties may be used to syntactically prove type safety for this language [WF94].

[4] Because *size* operates over data-structures, the extension of *size* to polymorphic types is a little dubious. However, my observation that the *size* of all types is constant means that I can provide a branch for polymorphic types. I need to supply the *size* of the abstract type, no matter its identity; since this value is a constant over all types, I can just use the *size* of *int*.

$$
\begin{array}{l}
\mathit{fix}\ \mathit{copy} : \forall\alpha : \star . T(\alpha \to \alpha).\\
\quad \Lambda\alpha : \star .\ \mathit{typerec}[\alpha . T(\alpha \to \alpha)]\ \alpha\ \ \mathit{of}\\
\quad\quad \begin{array}{ll}
\mathit{int} & \Rightarrow \lambda i{:}\,\mathit{int}\,.i\\
\to & \Rightarrow \Lambda\alpha{:}\star.\lambda r_\alpha{:}T(\alpha\to\alpha).\Lambda\beta{:}\star.\lambda r_\beta{:}T(\beta\to\beta).\\
 & \quad \lambda f{:}T(\alpha\to\beta).r_\beta \circ f \circ r_\alpha\\
\times & \Rightarrow \Lambda\alpha{:}\star.\lambda r_\alpha{:}T(\alpha\to\alpha).\Lambda\beta{:}\star.\lambda r_\beta{:}T(\beta\to\beta).\\
 & \quad \lambda x{:}T(\alpha\times\beta).\langle r_\alpha(\pi_1 x), r_\beta(\pi_2 x)\rangle\\
\mu_\star & \Rightarrow \Lambda\alpha{:}\star\to\star.\lambda r{:}\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta).\\
 & \quad\quad \mathit{fix}\ f{:}T(\mu_\star\alpha\to\mu_\star\alpha).\lambda x{:}T(\mu_\star\alpha).\ \mathit{roll}\ (r\ [\mu_\star\alpha]\ f\ (\mathit{unroll}\, x))\\
\forall_\star & \Rightarrow \Lambda\alpha{:}\star\to\star.\lambda r{:}\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta).\\
 & \quad\quad \lambda x{:}T(\forall_\star\alpha).\Lambda\beta{:}\star.r\ [\beta]\ (\mathit{copy}\ [\beta])\,(x[\beta])\\
\exists_\star & \Rightarrow \Lambda\alpha{:}\star\to\star.\lambda r{:}\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta).\lambda x{:}T(\exists_\star\alpha).\\
 & \quad\quad \mathit{let}\langle\beta, y\rangle = \mathit{unpack}\, x\ \mathit{in}\ \ \mathit{pack}\langle\beta, r\ [\beta]\ (\mathit{copy}\ [\beta])\ \ y\rangle\ \mathit{as}\ \exists\beta{:}\star.\alpha\beta
\end{array}
\end{array}
$$

Figure 6.2: Example: *copy*

Lemma 6.2.1 (Substitution) *We must prove four properties:*

1. *If* $\Delta, \beta{:}\kappa; \Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho$ *and* $\Delta \vdash c : \kappa$ *then* $\Delta[\alpha.\sigma[c/\beta]]; \Gamma[c/\beta] \vdash \Delta' \mid \eta[c/\beta] \mid \rho[c/\beta]$.
2. *If* $\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$ *and* $\Delta \vdash c : \kappa$ *then* $\Delta; \Gamma[c/\alpha] \vdash e[c/\alpha] : \sigma[c/\alpha]$.
3. *If* $\Delta; \Gamma, x{:}\sigma'[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho$ *and* $\Delta; \Gamma \vdash e' : \sigma'$ *then* $\Delta; \Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta[e'/x] \mid \rho$
4. *If* $\Delta; \Gamma, x{:}\sigma' \vdash e : \sigma$ *and* $\Delta; \Gamma \vdash e' : \sigma'$ *then* $\Delta; \Gamma \vdash e[e'/x]$

Proof

We prove the first two parts by simultaneous induction on the derivations

$$\Delta, \beta{:}\kappa; \Gamma\ [\alpha.\sigma] \vdash \Delta' \mid \eta\ \mid \rho \text{ and } \Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$$

with a case analysis on the last step. Because part two is similar to the substitution lemma for LI, we only include the case for *typerec*. The proofs of the second two parts are by simultaneous induction on the derivations

$$\Delta; \Gamma, x{:}\sigma'\ [\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho \text{ and } \Delta; \Gamma, x{:}\sigma' \vdash e : \sigma$$

Those proofs follow analogously to the proofs of the first two parts, so we do not detail them here.

1. In the base case,
$$\overline{\Delta;\Gamma[c/\beta][\alpha.\sigma] \vdash \emptyset \mid \emptyset \mid \emptyset}$$

   Otherwise, assume

$$\begin{array}{ll} \Delta,\beta{:}\kappa;\Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho & \Delta,\beta{:}\kappa;\Gamma \vdash c' : \kappa' \\ \Delta,\beta{:}\kappa;\Gamma \vdash e : [\alpha.\sigma]\langle c' : \kappa'\rangle & \gamma \notin \mathit{Dom}(\Delta,\Delta') \end{array}$$

$$\begin{array}{ll} \text{By induction} & \Delta;\Gamma[c/\beta][\alpha.\sigma] \vdash \Delta' \mid \eta[c/\beta] \mid \rho[c/\beta] \\ \text{By Lemma 2.5.3} & \Delta \vdash c'[c/\beta] : \kappa' \\ \text{By Part 2} & \Delta;\Gamma[c/\beta] \vdash e[c/\beta] : ([\alpha.\sigma]\langle c : \kappa'\rangle)[c/\beta] \end{array}$$

   Therefore we may conclude

$$\Delta;\Gamma[c/\beta][\alpha.\sigma[c/\beta]] \vdash \Delta',\gamma{:}\kappa \mid (\eta,\gamma{:}e)[c/\beta] \mid (\rho,\gamma{:}c')[c/\beta]$$

2. Assume the last rule of the derivation was:

$$\frac{\begin{array}{c} \Delta,\beta{:}\kappa,\alpha{:}\star \vdash \sigma \\ \Delta,\beta{:}\kappa;\Gamma[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho \\ \Delta,\beta{:}\kappa,\Delta' \vdash c' : \kappa' \\ \Delta,\beta{:}\kappa;\Gamma \vdash e_\oplus : [\alpha.\sigma]\langle \oplus : \kappa_\oplus\rangle \qquad (\forall e_\oplus \in \overline{e}) \end{array}}{\Delta,\beta{:}\kappa;\Gamma \vdash \mathit{typerec}[\alpha.\sigma][\Delta',\eta,\rho]\, c' \;\mathit{of}\; \overline{e} : [\alpha.\sigma]\langle \rho(c') : \kappa'\rangle}$$

$$\begin{array}{ll} \text{By Lemma 2.5.3} & \Delta,\alpha{:}\star \vdash \sigma[c/\beta] \\ \text{By Part 1} & \Delta;\Gamma[c/\beta][\alpha.\sigma[c/\beta]] \vdash \Delta' \mid \eta[c/\beta] \mid \rho[c/\beta] \\ \text{By Lemma 2.5.3} & \Delta,\Delta' \vdash c'[c/\beta] : \kappa' \\ \text{By induction} & \Delta;\Gamma[c/\beta] \vdash e_\oplus[c/\beta] : ([\alpha.\sigma]\langle \oplus : \kappa_\oplus\rangle)[c/\beta] \;\; (\forall e_\oplus \in \overline{e}) \end{array}$$

   Therefore we may conclude

$$\Delta;\Gamma[c/\beta] \vdash (\mathit{typerec}[\alpha.\sigma][\Delta',\eta,\rho]\, c' \;\mathit{of}\; \overline{e})[c/\beta] : ([\alpha.\sigma]\langle \rho(c') : \kappa'\rangle)[c/\beta]$$

□

Lemma 6.2.2 (Subject Reduction) *If* $\vdash e : \sigma$ *and* $e \mapsto e'$ *then* $\vdash e' : \sigma$.

Proof

Again proof is by induction on the derivation $\vdash e : \sigma$. We only show the case involving *typerec*. Assume the last rule was

$$\frac{\begin{array}{c}\alpha{:}\star \vdash \sigma \\ \alpha.\sigma \vdash \Delta' \mid \eta \mid \rho \\ \Delta' \vdash c : \kappa \\ \vdash e_{\oplus} : [\alpha.\sigma]\langle \oplus : \kappa_{\oplus} \rangle \qquad (\forall e_{\oplus} \in \overline{e})\end{array}}{\vdash \mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\, c \;\mathit{of}\; \overline{e} : [\alpha.\sigma]\langle \rho(c) : \kappa \rangle}$$

case $\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\, \beta \;\mathit{of}\; \overline{e} \mapsto \eta(\beta)$

By assumption, $\vdash \eta(\beta) : [\alpha.\sigma]\langle \rho(\beta) : \Delta(\beta) \rangle$.

case $\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho] \;\oplus\; \mathit{of}\; \overline{e} \mapsto e_{\oplus}$

By assumption, $\vdash e_{\oplus} : [\alpha.\sigma]\langle \oplus : \kappa_{\oplus} \rangle$.

case $\kappa = \kappa_1 \to \kappa_2$ and

$$\begin{array}{l}\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\, (\lambda\beta{:}\kappa_1.c_1) \;\mathit{of}\; \overline{e} \\ \quad \mapsto \Lambda\gamma{:}\kappa_1.\lambda x{:}[\alpha.\sigma]\langle \gamma : \kappa_1 \rangle. \\ \qquad \mathit{typerec}[\alpha.\sigma][\Delta', \beta{:}\kappa_1, \eta, \beta{:}x, \rho, \beta{:}\gamma]\; c \;\mathit{of}\; \overline{e}.\end{array}$$

We wish to show that

$$\begin{array}{l}\gamma{:}\kappa_1; x{:}[\alpha.\sigma]\langle \gamma : \kappa_1 \rangle \\ \quad \vdash \mathit{typerec}[\alpha.\sigma][\Delta', \beta{:}\kappa_1, \eta, \beta{:}x, \rho, \beta{:}\gamma]\; c \;\mathit{of}\; \overline{e} \\ \qquad : [\alpha.\sigma]\langle \rho, \beta{:}\gamma(c_1) : \kappa_2 \rangle\end{array}$$

This follows as $\Delta', \beta{:}\kappa_1 \vdash c_1 : \kappa_2$ and as

$$\frac{\begin{array}{c}[\alpha.\sigma] \vdash \Delta' \mid \eta \mid \rho \\ \gamma{:}\kappa_1 \vdash \gamma : \kappa_1 \\ \gamma{:}\kappa_1;\; x{:}([\alpha.\sigma]\langle \gamma : \kappa_1 \rangle) \vdash x : [\alpha.\sigma]\langle \gamma : \kappa_1 \rangle\end{array}}{\gamma{:}\kappa_1; x{:}([\alpha.\sigma]\langle \gamma : \kappa_1 \rangle)\; [\alpha.\sigma] \vdash \Delta', \beta{:}\kappa_1 \mid \eta, \beta{:}x \mid \rho, \beta{:}\gamma}$$

case The argument to *typerec* is an application $c_1 c_2$.

$$\begin{array}{l}\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\; (c_1 c_2) \;\mathit{of}\; \overline{e} \mapsto \\ \quad (\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\; c_1 \;\mathit{of}\; \overline{e})[\rho(c_2)](\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\; c_2 \;\mathit{of}\; \overline{e})\end{array}$$

We wish to show

$$\begin{array}{l}\vdash (\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\; c_1 \;\mathit{of}\; \overline{e})[\rho(c_2)](\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\; c_2 \;\mathit{of}\; \overline{e}) \\ \quad : [\alpha.\sigma]\langle \rho(c_1 c_2) : \kappa \rangle\end{array}$$

Suppose $\Delta' \vdash c_1 : \kappa' \to \kappa$. The above follows from the following three judgments:

1. $\vdash (\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\ c_1\ \ \mathit{of}\ \ \bar{e}) : [\alpha.\sigma]\langle c_1 : \kappa' \to \kappa\rangle$
   This result follows from the preconditions of *typerec*.
2. $\vdash \rho(c_2) : \kappa'$
   This result follows by substitution, as $\Delta' \vdash c_2 : \kappa'$
3. $\vdash (\mathit{typerec}[\alpha.\sigma][\Delta', \eta, \rho]\ c_2\ \ \mathit{of}\ \ \bar{e}) : [\alpha.\sigma]\langle c_2 : \kappa'\rangle$
   This result follows from the preconditions of *typerec*.

□

Lemma 6.2.3 (Progress) *If* $\vdash e : \sigma$ *and* e *is not a value then there exists an* e' *such that* $e \mapsto e'$*.*

As long as every well-typed *typerec* expression steps, the Progress Lemma follows from that of LI. This fact is true because for each form of constructor argument to *typerec* (variable, application, abstraction, or operator), there is a rule of the operational semantics.

Theorem 6.2.4 (Type Soundness) *If* $\emptyset \vdash e : \sigma$ *and* $e \mapsto^* e'$ *then* e' *is not stuck.*

Proof

See Theorem 2.5.8. □

6.2.4 Model theoretic properties

Furthermore, we would like to make precise the notion that the term language interprets the type language. In order to do so, we must define an appropriate notion of equality for the term language, so that we can prove that equality in the type language is preserved by equality in the term language. What should this equivalence relations between terms be?

As is typical for programming language, the only relationship given outright between terms is evaluation (besides syntactic equality modulo α-conversion, of course). We could extend this relation to an equivalence relation as follows

$$e \equiv_{\mapsto} e' \stackrel{\text{def}}{=} e \mapsto^* v \text{ and } e' \mapsto^* v$$

However, this equivalence is too fine. Two functions may not be considered equal even if they are extensionally equal—for every pair of equivalent arguments they produce equivalent results. Therefore, we need to extend this notion of equivalence to include this idea of extensionality.

To do so, we define kind-indexed relations $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{C}[\![\cdot]\!]$ below. We will use $\mathcal{C}[\![\cdot]\!]$ to define what we mean by equivalence. However, the entire relation we define will not quite be an equivalence relation, only portion of it restricted to terms that terminate with values. We will consider non-terminating terms to be related to every other term.

Definition 6.2.5

$$\begin{array}{lcl}
\mathcal{V}[\![\star]\!] & = & \{(v,v) \;\;| \emptyset \vdash v : [c]\langle c' : \star\rangle\} \\
\mathcal{V}[\![\kappa_1 \to \kappa_2]\!] & = & \{(v_1,v_2)| \emptyset \vdash v : [c]\langle c' : \kappa_1 \to \kappa_2\rangle \textit{ and } \emptyset \vdash v_2 : [c]\langle c' : \kappa_1 \to \kappa_2\rangle \\
& & \quad \textit{for all } (v_1',v_2') \in \mathcal{V}[\![\kappa_1]\!] \textit{ for all } \emptyset \vdash c_1 = c_2 : \kappa_1, \\
& & \quad (v_1[c_1]v_1', v_2[c_2]v_2') \in \mathcal{C}[\![\kappa_2]\!] \;\} \\
\\
\mathcal{C}[\![\kappa]\!] & = & \{(e,e') \;\;| \emptyset \vdash e : [c]\langle c' : \kappa\rangle \textit{ and } \emptyset \vdash e' : [c]\langle c' : \kappa\rangle \\
& & \quad e \mapsto^* v \;\&\; e' \mapsto^* v' \& \; (v,v') \in \mathcal{V}[\![\kappa]\!] \;\}
\end{array}$$

Definition 6.2.6 *Define* $e \approx_{\mathcal{C}} e'$ *if either* $(e,e') \in \mathcal{C}[\![\kappa]\!]$ *or* e *diverges or* e' *diverges.*

Proposition 6.2.7 *We observe (without proof) a few trivial properties about these relations:*

1. $\mathcal{V}[\![\kappa]\!]$ *is an equivalence relation on closed well-typed terms.*
2. $\mathcal{C}[\![\kappa]\!]$ *is an equivalence relation on closed well-typed terms.*
3. *If* $(v,v') \in \mathcal{V}[\![\kappa]\!]$*, then* $(v,v') \in \mathcal{C}[\![\kappa]\!]$*.*
4. *If* $e \mapsto^* e'$*, then* $(e,e') \in \mathcal{C}[\![\kappa]\!]$*.*
5. *If* e *diverges, then* $e \approx_{\mathcal{C}} e'$ *for all* e'*.*

Now that we have a version of equivalence, we may define a collection of Henkin models for the type language, using the closed, well-typed portion of the term language.

For any c such that $\emptyset \vdash c : \star \to \star$, and for any $\overline{e}$ such that $e_\oplus \in [c]\langle \oplus : \kappa_\oplus\rangle$, we define the following *typed applicative structure* (also called a pre-frame) $\mathcal{A}_{c,\overline{e}} = (A^\kappa, \mathbf{App}^{\kappa_1,\kappa_2}, \mathit{Const})$:

- A^κ is $\{\ [e]_{\mathcal{C}[\![\kappa]\!]} \mid \emptyset \vdash e : [c]\langle c' : \kappa\rangle$ for some $\emptyset \vdash c' : \kappa\} \cup \{\bot\}$
- $\mathbf{App}^{\kappa_1,\kappa_2}\ e_1\ e_2$ is $e_1[c_2]e_2$, when $\emptyset \vdash e_2[c]\langle c_2 : \kappa_2\rangle$.
- $\mathit{Const}\ \oplus$ is $e_\oplus$

We must show that such an applicative structure satisfies the property of extensionality :

$$\text{For all } f, g \in A^{\kappa_1 \to \kappa_2},\ \text{if } \mathbf{App} f\ d\ \approx_{\mathcal{C}} \mathbf{App}\ g\ d\ \text{ for all } d \in A^{\kappa_1},\ \text{then } f \approx_{\mathcal{C}} g$$

We can show this property by unwinding the definitions. Suppose f, g are in $A^{\kappa_1 \to \kappa_2}$. If either f or g diverge, the property is trivially true. Otherwise, if $f \mapsto^* v_f$ and $g \mapsto^* v_g$. We want to show that $(v_f, v_g) \in \mathcal{V}[\![\kappa_1 \to \kappa_2]\!]$. This is true if, for all $(d_1, d_2) \in \mathcal{V}[\![\kappa_1]\!]$, $(v_f[c]d_1, v_g[d]d_2) \in \mathcal{C}[\![\kappa_2]\!]$. As $(d_1, d_2) \in \mathcal{V}[\![k_1]\!]$, $d_1 \approx_{\mathcal{C}} d_2$. Therefore $f[c]d_1 \approx_{\mathcal{C}} f[c]d_2$ which implies that $(v_f[c]d_1, v_g[c]d_2) \in \mathcal{C}[\![\kappa_2]\!]$.

Finally we may extend our typed applicative structure to an environment model by supplying a meaning function $\mathcal{A}_{c,\overline{e}}[\![\cdot]\!]\cdot$ from kinding derivations and environments, to elements of the model. This meaning function is defined only over environments that satisfy the type context Δ (written $\eta \vdash \Delta$). We define that notion using the judgment we have previously defined.

$$\eta \vdash \Delta \text{ if and only if there exists a } \rho \text{ such that } c \vdash \Delta \mid \eta \mid \rho$$

Now we may define the meaning function using *typerec*, and this ρ:

$$\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash c' : \kappa]\!]\eta = \mathit{typerec}[\Delta, \eta, \rho]\ c\ \overline{e}$$

Our model satisfies the *environment model condition* if the meaning function $\mathcal{A}_{c,\overline{e}}[\![\cdot]\!]\cdot$ satisfies the following properties.

$$\begin{array}{ll}
\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash \oplus : \kappa]\!]\eta & \approx_{\mathcal{C}} \mathit{Const}\ \oplus \\
\mathcal{A}_{c,\overline{e}}[\![\Delta, \alpha{:}\kappa \vdash \alpha : \kappa]\!]\eta & \approx_{\mathcal{C}} \eta(\alpha) \\
\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash c_1 c_2 : \kappa]\!]\eta & \approx_{\mathcal{C}} \mathbf{App}(\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash c_1 : \kappa_1 \to \kappa_2]\!]\eta) \\
 & \quad (\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash c_2 : \kappa_1]\!]\eta) \\
\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash \lambda\alpha{:}\kappa.c : \kappa_1 \to \kappa]\!]\eta & \approx_{\mathcal{C}} \text{the unique } f \text{ such that } \forall d \in A^{\kappa_1}, \\
 & \quad \mathbf{App}\ fd \approx_{\mathcal{C}} \mathcal{A}_{c,\overline{e}}[\![\Delta, \alpha{:}\kappa \vdash c : \kappa_2]\!]\eta, \alpha{:}d
\end{array}$$

The first three properties follow trivially from the definition. Consider the fourth property, we must show that: $\forall d \in A^{\kappa_1}$

$$\mathbf{App}\ \mathcal{A}_{c,\overline{e}}[\![\Delta \vdash \lambda\alpha{:}\kappa.c : \kappa_1 \to \kappa]\!]\eta\ d\ \approx_{\mathcal{C}}\ \mathcal{A}_{c,\overline{e}}[\![\Delta, \alpha{:}\kappa \vdash c : \kappa_2]\!]\eta, \alpha{:}d$$

This proposition reduces to

$$(\mathit{typerec}[\Delta, \eta, \rho](\lambda\alpha{:}\kappa.c)\ \overline{e})\ [c']\ d\ \approx_{\mathcal{C}}\ \mathit{typerec}[\Delta, \alpha{:}\kappa, \eta, \alpha{:}d, \rho, \alpha{:}c']\ c\ \overline{e}$$

(where $\emptyset \vdash d : [c]\langle c' : \kappa_1 \rangle$) which follows immediately. Because uniqueness follows from extensionality, we are done.

A number of important properties hold for environment models of the simply typed lambda calculus. The most important of these is that equality is preserved by the model.

Table 6.3: LH: Semantics for multiplace *typerec*

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{\begin{array}{l}\Gamma; c \vdash \Gamma' \mid \eta \mid \rho_1 \mid \ldots \mid \rho_n \\ \Gamma, \Gamma' \vdash c' : \kappa \qquad \Gamma \vdash c : \star^n \to \star \\ \Gamma \vdash e_\oplus : c\langle\kappa_\oplus\rangle^n \oplus \ldots \oplus \qquad (\forall e_\oplus \in \overline{e})\end{array}}{\Gamma \vdash \mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1 \ldots \rho_n]\, e \ \mathit{of}\ \ \overline{e} : c\langle\kappa\rangle^n \rho_1(c') \ldots \rho_n(c')}$$

$\boxed{e \mapsto e'}$

$$\mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1, \ldots, \rho_n] \ \ \oplus \ \ \mathit{of} \ \ \overline{e} \ \mapsto \eta_\oplus$$

$$\mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1, \ldots, \rho_n] \ \alpha \ \ \mathit{of} \ \ \overline{e} \ \mapsto \eta(\alpha)$$

$$\begin{array}{l}\mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1, \ldots, \rho_n] \ (c_1 c_2) \ \mathit{of} \ \ \overline{e} \\ \mapsto (\mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1, \ldots, \rho_n] \ c_1 \ \mathit{of} \ \ \overline{e}) \ [\rho_1(c_2)] \ldots [\rho_n(c_2)] \\ \quad (\mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1, \ldots, \rho_n] \ c_2 \ \mathit{of} \ \ \overline{e})\end{array}$$

$$\begin{array}{l}\mathit{typerec}^n[\alpha.\sigma][\Gamma', \eta, \rho_1, \ldots, \rho_n] \ (\lambda\alpha{:}\kappa.c') \ \ \mathit{of} \ \ \overline{e} \\ \mapsto \Lambda\beta_1{:}\kappa. \ldots \Lambda\beta_n{:}\kappa.\lambda x{:}c\langle\kappa\rangle^n \beta_1 \ldots \beta_n. \\ \quad (\mathit{typerec}^n[\alpha.\sigma][\Gamma', \alpha{:}\kappa, \eta, \alpha{:}x, \rho_1, \alpha{:}\beta_1, \ldots, \rho_n, \alpha{:}\beta_n] \ c' \ \ \mathit{of} \ \ \overline{e})\end{array}$$

Corollary 6.2.8 *(Soundness) If $\Delta \vdash c = c' : \kappa$ then for any $\eta \vdash \Delta$,*

$$\mathcal{A}_{c,\overline{e}}[\![\Delta \vdash c : \kappa]\!]\eta \approx_{\mathcal{C}} \mathcal{A}_{c,\overline{e}}[\![\Delta \vdash c' : \kappa]\!]\eta.$$

This property means that we can change the argument to *typerec* to an equivalent constructor at any point and the term will still evaluate roughly the same. There is an issue with non-termination—because our term equality equated non-termination with any term, it is possible for a *typerec* over one constructor to diverge, and one over an equivalent constructor not to. However, by moving to a call-by-name operational semantics, we may avoid this problem.

6.3 Multiplace logical relations

The definition of polykinded types $[\alpha.\sigma]\langle c' : \kappa\rangle$ follows the definition of a *unary logical relation* over type constructors indexed by the kind κ. In order to write some polytypic functions (such as map and zip), Hinze observed that I need logical relations over multiple type constructors. To support multiplace relations in this framework, we generalize $[\alpha.\sigma]\langle\!|c' : \kappa\rangle$. For an n-place relation, c must take n arguments, each of kind $\star$. I abbreviate c's kind as $\star^n \to \star$.

$$c\langle\star\rangle^n c_1 \ldots c_n = T(c\ c_1 \ldots c_n)$$
$$c\langle\kappa_1 \to \kappa_2\rangle^n c_1 \ldots c_n = \forall\beta_1{:}\kappa_1.\ldots\forall\beta_n{:}\kappa_1.c\langle\kappa_1\rangle^n\beta_1\ldots\beta_n \to c\langle\kappa_2\rangle^n(c_1\beta_1)\ldots(c_n\beta_n)$$

Changing this definition forces us to generalize *typerec*, expanding ρ to a set of type environments $\rho_1 \ldots \rho_n$, and extending the judgment $\Gamma; c \vdash \Gamma' \mid \eta \mid \rho$ as below. I use this set of type environments in the modified operational semantics to provide substitutions for the n type arguments in a type application. Furthermore, on type abstraction, n type variables are abstracted, and all environments, $\rho_1 \ldots \rho_n$, are extended with these variables (see Table 6.3).

$$\frac{}{\Gamma; c \vdash \emptyset \mid \emptyset \mid \emptyset_1 \mid \ldots \mid \emptyset_n}$$

$$\frac{\Gamma; c \vdash \Gamma' \mid \eta \mid \rho_1 \ldots \rho_n \qquad \Gamma \vdash c_1 : \kappa \quad \ldots \quad \Gamma \vdash c_n : \kappa \qquad \Gamma \vdash e : [\alpha.\sigma]\langle\!|c_1 \ldots c_n : \kappa\rangle \qquad \alpha \notin Dom(\Gamma, \Gamma')}{\Gamma; c \vdash \Gamma', \alpha{:}\kappa \mid \eta, \alpha{:}e \mid \rho_1, \alpha{:}c_1 \mid \ldots \mid \rho_n, \alpha{:}c_n}$$

6.3.1 Example: map

For example, a generalized version of the function map can be defined using *typerec*^{2}, with type $\forall\alpha{:}\star \to \star.(\to)\langle\star \to \star\rangle^2\alpha\ \alpha$. The definition of this function is essentially a two-place version of *copy*. If map is instantiated with the type constructor *list*, the result is the standard map over lists with type :
$(\to)\langle\star \to \star\rangle^2 list\ list = \forall\alpha{:}\star.\forall\beta{:}\star.(\alpha \to \beta) \to (list\ \alpha \to list\ \beta)$.

$$\begin{array}{ll}
\multicolumn{2}{l}{map = typerec[\lambda\alpha_1{:}\star.\lambda\alpha_2{:}\star.\alpha_1 \to \alpha_2]\ \alpha\ \ of} \\
int & \Rightarrow \lambda i{:}\, int\,.i \\
\to & \Rightarrow \text{undefined} \\
\times & \Rightarrow \Lambda\alpha_1, \alpha_2{:}\star.\lambda r_\alpha{:}T(\alpha_1 \to \alpha_2).\Lambda\beta_1, \beta_2{:}\star.\lambda r_\beta{:}T(\beta_1 \to \beta_2). \\
 & \quad \lambda x{:}T(\alpha_1 \times \beta_1).\langle r_\alpha(\pi_1 x), r_\beta(\pi_2 x)\rangle \\
\mu_\star & \Rightarrow \Lambda\alpha_1, \Lambda\alpha_2{:}\star \to \star.\lambda r{:}\forall\beta_1, \beta_2{:}\star.T(\beta_1 \to \beta_2) \to T(\alpha_1\beta_2 \to \alpha_2\beta_2). \\
 & \qquad fix\ f{:}T(\mu_\star\alpha_1 \to \mu_\star\alpha_2).\lambda x{:}T(\mu_\star\alpha_1).\ roll\ (r\ [\mu_\star\alpha_1][\mu\alpha_2]\ f\ (unroll\ x)) \\
\forall_\star & \Rightarrow \Lambda\alpha_1, \Lambda\alpha_2{:}\star \to \star.\lambda r{:}(\forall\beta_1, \beta_2{:}\star.T(\beta_1 \to \beta_2) \to T(\alpha_1\beta_1 \to \alpha_2\beta_2)). \\
 & \qquad \lambda x{:}T(\forall_\star\alpha_1).\Lambda\beta{:}\star.r[\beta][\beta]\,(\lambda y{:}\beta.y)\,(x[\beta]) \\
\exists_\star & \Rightarrow \Lambda\alpha_1, \alpha_2{:}\star \to \star.\lambda r{:}(\forall\beta_1, \beta_2{:}\star.T(\beta_1 \to \beta_2) \to T(\alpha_1\beta_1 \to \alpha_2\beta_2)). \\
 & \qquad \lambda x{:}T(\exists_\star\alpha_1). \\
 & \qquad\quad let\langle\beta, y\rangle = unpack\ x\ in\ \ pack\langle\beta, r[\beta][\beta]\ (\lambda z{:}\beta.z)\ y\rangle\ as\ \exists\beta{:}\star.\alpha\beta
\end{array}$$

Unlike *copy*, there is no *fix* surrounding *map* to provide a recursive call in the cases for $\forall_\star$ and $\exists_\star$. The *typerec* that comprises *map* when applied to a constructor of kind $\star$ is an identity function, so it makes sense that in each of these branches r is called with an identity function.

6.3.2 Example: typetostring

A surprising observation is that there are useful functions when $n = 0$, such as *typetostring* below. In this code, *gensym* creates a unique string for each variable name, and $let\ \ x = e_1\ in\ \ e_2$ is the usual abbreviation for $(\lambda x{:}\sigma.e_2)e_1$.

$$
\begin{array}{l}
typetostring : \forall\alpha : \star.\ string\ . \\
typetostring = \Lambda\alpha : \star.\ typerec^0[string]\ \alpha\ of \\
\quad \begin{array}{ll}
int & \Rightarrow \texttt{"int"} \\
\rightarrow & \Rightarrow \lambda x : string\ .\lambda y : string\ .\texttt{"("}x \mathbin{++} \texttt{" -> "} \mathbin{++} y \mathbin{++} \texttt{")"} \\
\times & \Rightarrow \lambda x : string\ .\lambda y : string\ .\texttt{"("}x \mathbin{++} \texttt{" * "} \mathbin{++} y \mathbin{++} \texttt{")"} \\
\mu_\star & \Rightarrow \lambda r : string \rightarrow string\ . \\
 & \quad let\ x = gensym()\ in\ \texttt{"mu"} \mathbin{++} x \mathbin{++} \texttt{"."} \mathbin{++} (rx) \\
\forall_\star & \Rightarrow \lambda r : string \rightarrow string\ . \\
 & \quad let\ x = gensym()\ in\ \texttt{"all"} \mathbin{++} x \mathbin{++} \texttt{"."} \mathbin{++} (rx) \\
\exists_\star & \Rightarrow \lambda r : string \rightarrow string\ . \\
 & \quad let\ x = gensym()\ in\ \texttt{"ex"} \mathbin{++} x \mathbin{++} \texttt{"."} \mathbin{++} (rx)
\end{array}
\end{array}
$$

Note, this example does not follow the pattern of iso-recursive types, which would be $\mu_\star \Rightarrow \lambda r{:}\, string \rightarrow string\,.\,fix\ f{:}\, string\,.rf$. In that case, the string representation of a recursive type would be infinitely long, witnessing the fact that a recursive type is an infinitely large type. I could have also written this code using $typerec^1$, but it would have been clumsy in the branches for quantified types. In these branches, r would be of type $\forall\alpha{:}\star.\ string \rightarrow string$ instead of $string \rightarrow string$ as above, so a dummy type argument must be supplied when r is used.

6.4 Kind polymorphism

Why is there a distinction between types σ, and type constructors c, necessitating the irritating conversion $T(c)$? The reason is that not all types are analyzable. In particular, we cannot analyze polymorphic types where the kind of the bound variable is not $\star$, only those types created with the constructor $\forall_\star$. Trifonov et al.[TSS00] (hereafter TSS) use the term *fully reflexive* to refer to a calculus where

Table 6.4: LH: Additions for kind polymorphism

κ	::=	$\ldots \mid \chi \mid \forall\chi.\kappa$
$\oplus$	::=	$\ldots \mid \forall \mid \exists \mid \forall^{+}$
c	::=	$\ldots \mid \Lambda\chi.c \mid c[\kappa] \mid c\langle\kappa\rangle^{n} c_1 \ldots c_n$
σ	::=	$\ldots \mid \forall^{+}\chi.\sigma$
e	::=	$\ldots \mid \Lambda^{+}\chi.e \mid e[\kappa]^{+}$

analysis operations are applicable to all types, and argue that this property is important for a type analyzing language.

A naive idea to make this language fully reflexive would be to limit polymorphism to that of F2, i.e., allow polymorphic types only of the form $\forall\alpha{:}\star.\sigma$. However, then I cannot express the type of the $e_{\forall_\star}$ branch as it quantifies over a constructor of kind $\star \to \star$. I could then extend the language to allow types that quantify over constructors of kind $\star \to \star$, and add a constructor ($\forall_3$) of kind $((\star \to \star) \to \star) \to \star$, but then the $e_{\forall_3}$ branch would quantify over variables of kind $(\star \to \star) \to \star$. In general, I have a vicious cycle: for each type that I add to the calculus, I need a more complicated type to describe its branch in *typerec*. I could break this cycle by adding an infinite number of type constructors $\forall_\kappa$, thereby allowing construction of all polymorphic types. However, then *typerec* would require an infinite number of branches to cover all such types.

TSS avoid having an infinite number of branches for polymorphic types by introducing *kind polymorphism* in their type-analyzing language LP. By holding the kind of the bound variable abstract, they are able to write one branch for all such types. Furthermore, kind polymorphism is necessary in their calculus to analyze polymorphic types. As their analysis is based on induction over the kind $\star$, they cannot handle $\forall_\star$with a negative occurrence of $\star$ in the kind of its argument. With kind polymorphism, the $\forall$ constructor has kind $\forall\chi.(\chi \to \star) \to \star$, without such a negative occurrence.

The LH version of *typerec*, as it is not based on induction, can already analyze $\forall_\star$. So their second motivation for kind polymorphism does not apply. However, in this system with kind-indexed types, I do have a separate and additional reason for adding kind polymorphism—the higher-order *typerec* term is naturally kind polymorphic and I would like to express that fact in the type system.

Like TSS, I include two forms of kind polymorphism: As in the LU language, I extend the type constructor language to F2, by adding kind variables χ and polymorphic kinds $\forall\chi.\kappa$, and adding type constructors supporting kind abstraction ($\Lambda\chi.c$) and application $c[\kappa]$. This polymorphism allows us to express the kind of the $\forall$ and $\exists$ constructors as $\forall\chi.(\chi \to \star) \to \star$. Next, I also allow terms to abstract ($\forall^+\chi.e$) and apply ($e[\kappa]^+$) kinds, so that the $\forall$ and $\exists$ branches of *typerec* may be polymorphic over the domain kind. Furthermore, I introduce a new constructor $\forall^+$ to describe the type of kind-polymorphic terms. This constructor is also represented with higher-order abstract syntax: it is of kind $(\forall\chi.\star) \to \star$, where its argument describes how the type depends on the abstract kind χ.

To extend type analysis to polymorphic kinds I must extend the definition of $[\alpha.\sigma]\langle\!\langle \alpha : \kappa \rangle$ for the new kind forms χ and $\forall\chi.\kappa$. Therefore, the type constructor language now contains polykinded types and the following axioms to the equality judgment for type constructors, including one to deal with polymorphic kinds :

$$\frac{}{\Delta \vdash c\langle\star\rangle^n c_1 \ldots c_n = cc_1 \ldots c_n : \star}$$

$$\frac{}{\begin{array}{l}\Delta \vdash c\langle\kappa_1 \to \kappa_2\rangle^n c_1 \ldots c_n = \\ \quad \forall[\kappa_1](\lambda\alpha_1{:}\kappa_1. \ldots \forall[\kappa_1](\lambda\alpha_n{:}k_1.c\langle\kappa_1\rangle^n \alpha_1 \ldots \alpha_n \to c\langle\kappa_2\rangle^n (c_1\alpha_1) \ldots (c_n\alpha_n)) \ldots) : \star\end{array}}$$

$$\frac{}{\Delta \vdash c\langle\forall\chi.\kappa\rangle^n c_1 \ldots c_n = \forall^+(\Lambda\chi.c\langle\kappa\rangle^n (c_1[\chi]) \ldots (c_n[\chi])) : \star}$$

Furthermore, the operational semantics of *typerec* must cover arguments that are kind abstractions or kind applications. By the above definition, *typerec* must produce a kind polymorphic term when reaching a kind polymorphic constructor. Therefore, an argument to *typerec* of a polymorphic kind pushes the *typerec* through the kind abstraction. Likewise, when *typerec* reaches a kind application during analysis, it propagates the analysis through.

$$typerec^n[\alpha.\sigma][\Delta, \eta, \overline{\rho}]\ (\Lambda\chi.c)\ \ of\ \ \overline{e} \mapsto \Lambda^+\chi.\ typerec^n[\alpha.\sigma][\Delta, \eta, \overline{\rho}]\ (c[\chi])\ \ of\ \ \overline{e}$$

$$typerec^n[\alpha.\sigma][\Delta, \eta, \overline{\rho}]\ (c[\kappa])\ \ of\ \ \overline{e} \mapsto (typerec^n[\alpha.\sigma][\Delta, \eta, \overline{\rho}]\ c\ \ of\ \ \overline{e})[\kappa]^+$$

With kind polymorphism, we can express the type of *size* precisely.

$$\forall^+\chi.\forall\alpha{:}\chi.\,T([\lambda\beta{:}\star.\beta \to int]\langle\!\langle \alpha : \chi \rangle).$$

The definition of *size* can also extend *size* to general existential types. Before, as $\exists_\star$ only hides type constructors of kind $\star$, the constant zero function was the *size* of the hidden type. Here, because the hidden type constructor may be of any kind, this branch uses a recursive call to define *size* for the hidden type.

$$\begin{array}{ll}\exists & \Rightarrow \Lambda^+\chi.\Lambda\alpha{:}\chi \to \star.\lambda r{:}(\forall\beta{:}\chi.[\alpha.\sigma]\langle\!\langle \beta : \chi \rangle \to T(\alpha\beta \to int)). \\ & \quad \lambda x{:}T(\exists[\chi]\alpha). \\ & \quad let\ \langle\beta, y\rangle = unpack\,x\,in\ \ r\ [\beta]\ \ (size[\chi][\beta])\ \ y\end{array}$$

6.4.1 Analysis of polymorphic types

In Section 6.2 showed that the operation of higher-order *typerec* over product types mirrored LI's operational semantics. How does analysis of polymorphic and existential types differ when *typerec* is viewed as an induction over the structure of types, as in TSS, and when *typerec* is viewed as an interpretation of the type language?

In the first case, (which I will distinguish by $typerec^i$) we have the following operational rule for polymorphic types; when c' is analyzed, its argument β is also examined with the same analysis.

$$typerec^i[\alpha.\sigma]\ (\forall[\kappa]c')\ \ of\ \ \overline{e}\ \mapsto e_\forall\ [\kappa]^+[c']\ (\Lambda\beta{:}\kappa.\ typerec^i[\alpha.\sigma]\ (c'\beta)\ \ of\ \ \overline{e})$$

Alternatively, with higher-order *typerec*, derives the following rule for polymorphic types that is not identical to the rule above. In this case, the result of analysis of the argument to c' may be supplied in the term argument x.

$$\begin{array}{l} typerec[\alpha.\sigma][\Delta,\eta,\rho]\ (\forall[\kappa]c')\ \ of\ \ \overline{e}\ \mapsto^* \\ \quad e_\forall\ [\kappa]^+[c']\ (\Lambda\beta{:}\kappa.\lambda x{:}\,T([\alpha.\sigma]\langle\beta:\kappa\rangle). \\ \qquad\qquad\qquad typerec[\alpha.\sigma][\Delta,\alpha{:}\kappa,\eta,\alpha{:}x,\rho,\alpha{:}\beta]\ (c'\alpha)\ \ of\ \ \overline{e}) \end{array}$$

However, many examples of polytypic functions defined by higher-order *typerec* (such as *copy*) create a fixed point of the Λ-abstracted *typerec* term, and it is this fixed point applied to β that eventually supplied for x. In that case, as above, the argument to c' is examined with the same analysis.

So, besides the ability to define operations over higher-order type constructors (such as *size* and *map*), *typerec* in LH has additional expressiveness over $typerec^i$: more flexibility in the analysis of quantified types. Type analyzing operations in LH are not required to call themselves recursively on the hidden type variable in an existential package, or on the arbitrary type argument to a polymorphic term. For example, a serializer written in LH could keep abstract parts of a data structure hidden:

$$abstract\_tostring(pack\langle int,\langle 2,3\rangle\rangle\ as\ \exists\alpha{:}\star.\alpha\times int)$$

could return `"(<hidden object>, 3)"`.

This difference also shows up in the following example, which uses *typetostring* to demonstrate the limits of intensional type analysis. With $typerec^i$ it is impossible to implement a version of `typetostring` that it may display all types. LH can do a little better: as before, it may display polymorphic types. However, LH also runs into trouble with kind polymorphism.

6.4.2 Example: typetostring

Unfortunately, even though the constructor language is much more expressive, it is impossible to extend *typetostring* in LH to create strings of constructors of all kinds. As kind polymorphism is parametric, it cannot differentiate constructors with polymorphic kinds. However, by giving *typetostring* a kind-polymorphic type, it may extend type functions and to types formed with $\forall$.

$$typetostring : \forall^{+}\chi.\forall\alpha{:}\chi.\,T(string\langle\chi\rangle^{0})$$

How can *typetostring* produce strings of higher-order type constructors? When χ is not $\star$, the result of *typetostring* is not a *string*. However, it may analyze the result type $string\langle\chi\rangle^{0}$ to produce a string when χ is a function kind.

Using a technique similar to *type-directed partial evaluation* [Dan96] we may *reify* a term of type $string\langle\chi\rangle^{0}$ into a string. The functions *app* and *lam* are necessary to create string abstractions and applications.

$$\begin{array}{l} lam : (string \to string) \to string \\ lam = \lambda x{:}\, string \to string\,. \\ \quad let\, b = gensym()\ in \\ \quad\quad \text{"(}\mathtt{lambda}\text{"} +\!\!+ b +\!\!+ \text{"."} +\!\!+ (x b) +\!\!+ \text{")"} \end{array} \qquad \begin{array}{l} app : string \to string \to string \\ app = \lambda x{:}\, string\,.\lambda y{:}\, string\,. \\ \quad \text{"("} +\!\!+ x +\!\!+ \text{" "} +\!\!+ y +\!\!+ \text{")"} \end{array}$$

Below, let $c = \lambda\alpha{:}\star.(\alpha \to string) \times (string \to \alpha)$

$$\begin{array}{l} ReifyReflect = typerec[\alpha.\sigma]\ \alpha\ of \\ \quad string \Rightarrow \langle \lambda y{:}\, string\,.y, \lambda y{:}\, string\,.y\rangle \\ \quad \to \quad \Rightarrow \Lambda\alpha_1{:}\star.\lambda r_1{:}c\alpha_1.\Lambda\alpha_2{:}\star.\lambda r_2{:}c\alpha_2. \\ \qquad\qquad let \langle reify_1, reflect_1\rangle = r_1 \\ \qquad\qquad\quad \langle reify_2, reflect_2\rangle = r_2\ in \\ \qquad\qquad \langle \lambda y{:}\alpha_1 \to \alpha_2.\ lam(reify_2 \circ y \circ reflect_1), \\ \qquad\qquad\ \lambda y{:}\, string\,.reflect_2 \circ app\ y\ \circ reify_1\rangle \end{array}$$

The result of *reify*, the first component of *ReifyReflect* above, composed with *typetostring* is a string representation of the long $\beta\eta$-normal form of the type constructor. What if that constructor has a polymorphic kind? There is not a reasonable branch for *ReifyReflect* in the case of $string\langle\forall\chi.\kappa\rangle^{0}$ because parametric kind polymorphism prevents us from writing analogous functions $klam : (\forall^{+}\chi.\, string) \to string$ and $kapp : string \to \forall\chi^{+}.\, string$. If the argument to *typetostring* is a kind polymorphic constructor, the best that we can do is return a constant string.

ReifyReflect is also necessary to create string representations of polymorphic types. In the previous version of *typetostring*, for the constructor $\forall_{\star}$, the inductive argument r was of type $string \to string$. With kind polymorphism, the type of the argument to r ($T(string\langle\chi\rangle^0)$) is dependent on χ the kind abstracted by $\forall$. In order to call r, we need to manufacture a value of this type–we need to *reflect* a string into the appropriate argument for the inductive call in *typetostring*:

$$\begin{array}{l} \forall \;\Rightarrow\; \Lambda^+\chi.\;\; \Lambda\alpha{:}\chi \to \star.\;\; \lambda r{:}\,T(string\langle\chi\rangle^0) \;\to\; string\,. \\ \quad\quad let\;\; \langle reify, reflect\rangle = ReifyReflect[string\langle\chi\rangle^0] \\ \quad\quad\quad\quad v = gensym\;\;()\;\; in\;\;\; \texttt{"all"} + v + \;\; \texttt{"."} + (r\;\;(reflect\;\; v)) \end{array}$$

Again, because *ReifyReflect* is limited to kinds of the form $\star$ or $\kappa_1 \to \kappa_2$, it can only print the polymorphic types of F_ω(i.e., types such as $\forall[\star \to \star](\lambda\alpha{:}\star \to \star.c)$, but not $\forall[\forall\chi.\kappa](\lambda\alpha{:}\forall\chi.\kappa.c)$). And just as there is not extension of *ReifyReflect* to kind-polymorphic constructors, there is no extension of *typetostring* to kind-polymorphic types (those formed by $\forall^+$).

Is this calculus fully reflexive? Yes. The types of this language are isomorphic to the constructors of kind $\star$, so there is no reason not to combine the two syntactic categories of type and type constructor. What this example shows is there is another property that we would like our systems to possess. In order to be able to write *typetostring* is should be possible for *typerec* to fully discriminate between all types. What this property means is that it is possible in this language to write a program that produces a different value for every type argument. An example of such a program is *typetostring* in LI.

There is a trade-off to be made. Either a calculus can be fully reflexive, or it can be fully discriminative. The previous non-example gives an informal justification that the kind-polymorphic calculus is not fully discriminative. Furthermore, TSS's language is also not fully discriminative.

However, kind polymorphism is not entirely to blame. If LH had added a predicative variant of kind-polymorphism, preventing type constructors such as $\forall[\Lambda\chi.\kappa]c$, and had not included $\forall^+$ in the type constructor language, the language would again be fully-discriminative. However, without the $\forall^+$ constructor the calculus would also not be fully reflexive.

6.5 Related work

In lifting type analysis to higher-order constructors, this work is related to work on induction over datatypes with embedded function spaces and more specifically to those datatypes representing higher-order abstract syntax. Meijer and Hutton [MH95] describe how to extend catamorphisms to datatypes with embedded functions by simultaneously computing an inverse (an *anamorphism*). Fegaras and Sheard [FS96] employ a different technique, noting that when the analyzed function is *parametric*, an inverse is not required. TSS employ this technique for the type level analysis of recursive types in the language LQ [TSS00], using a special kind to enforce that the argument to $\mu_\star$ is a parametric type function. Likewise, in a language for expressing induction over higher-order abstract syntax, Despeyroux et al. [DL01, DPC97], use a modal type discipline to indicate parametric functions. Because of the phase distinction between types and terms in the calculus of this paper, all analyzed type functions are parametric (as only terms analyze types) and so I do not require such additional typing machinery.

6.6 Chapter summary

In this chapter, I provide an operational semantics for type constructor polytypism, by extending *typerec* to cover higher-order types. By casting these operations in a type-passing framework, I may extend polytypic definitions over these type constructors (such as *size* and *map*) to languages where type abstraction cannot be specialized away at compile time. With type passing, I may also extend the domain of polytypic definitions to include first-class polymorphic and existential types, as I do for *size* and *typetostring*. With the addition of kind polymorphism and the inclusion of polykinded types as a form of type constructor, I allow the types of polytypic operations to be explicitly and accurately described. Finally, by extending *typerec* to constructors of polymorphic kind I allow the analysis of constructors such as $\forall$ and $\exists$ in a flexible manner.

Chapter 7

Representing higher-order type analysis

In the last chapter, I developed the calculus LH for analyzing higher-order and polymorphic type constructors with *typerec*. However, that calculus was an extension of LI and therefore did not possess the type-erasure semantics of LIR or any of its semantic benefits. The goal of this chapter is to develop a type-erasure version of higher-order type analysis.

To simplify the presentation, this chapter concentrates on the language of Section 6.2. It does not include the later extensions with multiplace *typerec* and kind polymorphism, although there are no technical restrictions to including these constructs in an erasure calculus.

7.1 Kind-directed execution: The LK language

In the previous chapter, in order to extend *typerec* to higher kinds, we defined its operational semantics as an interpreter of the type language. This interpretation operates syntactically. It maps type variables to term variables (using an environment), type abstractions to term abstractions and type applications to term applications.

The goal of this chapter is to change the process of interpreting the type language with the term language from run time to compile time. The process of phase splitting in Chapter 3 is also an interpretation of the constructor language with the term language. Again type abstraction is mapped to term abstraction, type application to term application and type variables to term variables.

In the type-erasure language, *typerec* must still interpret the term language to produce the correct result. There are problems with an analogous syntactic operational semantics for a phase-split version of the language. In Chapter 3, we defined the representation of a type constructor $\lambda\alpha{:}\kappa.c$ as $\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.\mathcal{R}|c|$. In the last chapter, the syntactic form of *typerec*'s argument determined the evaluation rule:

$$\textit{typerec}(\lambda\alpha{:}\kappa.c) \mapsto_h \ldots$$

In a type-erasure version, *typerec* would have to determine that its argument was a type abstraction surrounding a term abstraction.

$$\textit{typerec}(\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.e) \mapsto \ldots$$

However, this rule may not cover every representation of a type constructor. Evaluation of the representation argument to *typerec* may not always produce a syntactic λ as the subterm of the type abstraction. For example, the term

$$\Lambda\alpha{:}\kappa.((\lambda y.y)(\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.e))$$

is also well-typed as the representation of a type constructor. However, because evaluation will not reduce the application $((\lambda y.y)(\lambda x_\alpha{:}R\langle\alpha : \kappa\rangle.e))$ under the $\Lambda\alpha{:}\kappa$, this term will be stuck. Therefore, we will not be able to prove the Progress Lemma for this language. In most languages, including this one, evaluation is a process that is defined only over *closed terms*. We would have to greatly redefine what it means to evaluate expressions if we allowed a rule to reduce the body of a type abstraction.

As a precursor to the erasure version of the calculus, we first present an operational semantics for higher-order *typerec* that is directed by the *kind* of its argument as well as its syntax. This semantics does not examine the syntactic form of its argument when the argument is of higher kind. Therefore, we can phase split it into a type-erasure version in Section 7.3.

We call the LH language with this new operational semantics LK. Because this semantics is kind directed, we annotate the kind of the argument on the *typerec* term. Otherwise, there are no differences between the syntax and static semantics of LH and LK. Furthermore, this new operational semantics, though it may proceed in a different order than that of LH, will eventually produce the same value. Section 7.1.2 formalizes a proof of this fact.

Table 7.3 contains the LK operational semantics. Two relations define this semantics: the standard small step relation $\mapsto_k$ and a new relation $\Rightarrow_k$ for interpreting *typerec* when its argument is in a special form called a *path*. A path is a constructor that is in *weak-head normal form*, i.e., a constructor such that no weak-head reductions apply (see Table 7.1). It is not difficult to show that a path

Table 7.1: Weak-head reduction

$\boxed{c \rightsquigarrow^{wh} c'}$	
[*whr-β*]	$\dfrac{}{(\lambda\alpha{:}\kappa.c_1)c_2 \rightsquigarrow^{wh} c_1[c_2/\alpha]}$
[*whr-cong*]	$\dfrac{c_1 \rightsquigarrow^{wh} c_1'}{c_1 c_2 \rightsquigarrow^{wh} c_1' c_2}$

must be either an operator, a variable[1] or a path applied to another constructor:

$$p ::= \oplus \mid \alpha \mid p\ c$$

Table 7.2 describes the evaluation of a path. This evaluation is syntax directed, and the four rules are reminiscent of $(\mapsto_h)$ evaluation of *typerec* in the last chapter. Variables are interpreted by the environment and operators index the appropriate branches. Path evaluation continues through type application, fully expanding each application.

Path evaluation is used in the small-step operation of *typerec*. For arguments of kind $\star$, *typerec* first weak-head normalizes its argument and then employs path evaluation. For constructors of function kind (not necessarily a lambda term) path evaluation reduces the kind of its argument to *typerec* to a simpler kind by applying it to a new variable bound in *typerec*'s context.

7.1.1 Typing properties of LK

Like the operational semantics of the previous chapter, these new rules preserve the well-formedness of terms. In other words, if $\emptyset \vdash e : \sigma$ and $e \mapsto_k e'$ then $\emptyset \vdash e' : \sigma$. Furthermore, for any argument c to *typerec*, one of these rules applies.

Lemma 7.1.1 (Subject Reduction for LK) *Suppose*

$$\emptyset \vdash \mathit{typerec}[\kappa][\alpha.\sigma][\Delta, \eta, \rho]\ c\ \overline{e} : [\alpha.\sigma]\langle \rho(c) : \kappa \rangle.$$

1. *If c is a path and $\mathit{typerec}[\kappa][\alpha.\sigma][\Delta, \eta, \rho]\ c\ \overline{e} \Rightarrow_k e$ then $\emptyset \vdash e : [\alpha.\sigma]\langle \rho(c) : \kappa \rangle$.*
2. *If $\mathit{typerec}[\kappa][\alpha.\sigma][\Delta, \eta, \rho]\ c\ \overline{e} \mapsto_k e$ then $\emptyset \vdash e : [\alpha.\sigma]\langle \rho(c) : \kappa \rangle$.*

Proof

[1]During evaluation, this variable must be bound by an enclosing *typerec*.

Table 7.2: LK: Path evaluation

$\boxed{e \Rightarrow_k e'}$

[*pv-var*]
$$\frac{}{typerec[\kappa][\alpha.\sigma][\Delta,\eta,\rho]\ \beta\ \overline{e} \Rightarrow_k \eta(\beta)}$$

[*pv-const*]
$$\frac{}{typerec[\kappa][\alpha.\sigma][\Delta,\eta,\rho]\ \oplus\ \overline{e} \Rightarrow_k e_\oplus}$$

[*pv-app*]
$$\frac{\emptyset \vdash c_2 : \kappa_1 \qquad typerec[\kappa_1 \rightarrow \kappa][\alpha.\sigma][\Delta,\eta,\rho]\ c_1\ \overline{e} \Rightarrow_k e}{\begin{array}{l} typerec[\kappa][\alpha.\sigma][\Delta,\eta,\rho]\ (c_1 c_2)\ \overline{e} \\ \quad \Rightarrow_k e\ [\rho(c_2)]\ (typerec[\kappa_1][\alpha.\sigma][\Delta,\eta,\rho]\ c_2\ \overline{e}) \end{array}}$$

1. Proof by induction on the path c.
2. Proof by case analysis of κ.

□

Lemma 7.1.2 (Progress for LK) *Suppose*

$$\emptyset \vdash typerec[\kappa][\alpha.\sigma][\Delta,\eta,\rho]\ c\ \overline{e} : [\alpha.\sigma]\langle\rho(c) : \kappa\rangle$$

1. *If c is a path then there exists an e' such that $typerec[\kappa][\alpha.\sigma][\Delta,\eta,\rho]\ c\ \overline{e} \Rightarrow_k e'$.*
2. *There exists an e' such that $typerec[\kappa][\alpha.\sigma][\Delta,\eta,\rho]\ c\ \overline{e} \mapsto_k e'$.*

Proof

1. Proof by induction on the path c.
2. Proof by case analysis of κ.

□

7.1.2 Correspondence with LH

Next we show that this new kind-directed operational semantics evaluates in a manner that is similar to the that of the previous chapter. We can identify the *typerec* terms of LH and LK, as they only differ by the kind annotation. These languages have the same syntax and static semantics, but they have two different notions of evaluation.

Table 7.3: LK: Operational semantics

$$\boxed{e \mapsto_k e'}$$

[*ev-β*]

$$\frac{}{(\lambda x{:}\sigma.e)e' \mapsto e[e'/x]}$$

[*ev-app*]

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}$$

[*ev-ty-β*]

$$\frac{}{(\Lambda\alpha{:}\kappa.e)[c] \mapsto e[c/\alpha]}$$

[*ev-tapp*]

$$\frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

[*ev-trec-type*]

$$\frac{c \text{ weak-head normalizes to } p \qquad \mathit{typerec}[\star][\alpha.\sigma][\Delta,\eta,\rho]\ p\ \overline{e} \Rightarrow_k e}{\mathit{typerec}[\star][\alpha.\sigma][\Delta,\eta,\rho]\ c\ \overline{e} \mapsto_k e}$$

[*ev-trec-arrow*]

$$\frac{}{\begin{array}{l}\mathit{typerec}[\kappa_1 \to \kappa_2][\alpha.\sigma][\Delta,\eta,\rho]\ c\ \overline{e} \\ \quad \mapsto_k \Lambda\beta{:}\kappa_1.\ \lambda x_\beta{:}[\alpha.\sigma]\langle\beta : \kappa_1\rangle. \\ \qquad \mathit{typerec}[\kappa_2][\alpha.\sigma][\Delta,\gamma{:}\kappa_1,\eta,\gamma{:}x_\beta,\rho,\gamma{:}\beta]\ (c\gamma)\ \overline{e}\end{array}}$$

The following lemma states that path evaluation produces an equivalent term with respect to $\approx_{\mathcal{C}}$, the definition of equivalence of the last chapter.

Lemma 7.1.3 *For all* $\Delta \vdash p : \kappa$*, and* $\Delta'; \Gamma \vdash \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p\ \overline{e} : [c]\langle\!\langle \rho(p) : \kappa \rangle\!\rangle$

$$\text{if } \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p\ \overline{e} \Rightarrow_k e \text{ then } e \approx_{\mathcal{C}} \mathit{typerec}[\Delta, \eta, \rho]\ p\ \overline{e}.$$

Proof

By induction on $\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p\ \overline{e} \Rightarrow_k e$.

case (pv-var) $\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ \alpha\ \overline{e} \Rightarrow_k \eta(a)$ and $\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ \alpha\ \overline{e} \mapsto \eta(a)$ so

$$\eta(a) \approx_{\mathcal{C}} \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ \alpha\ \overline{e}$$

case (pv-const) Analogous to the variable case.
$\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ \oplus\ \overline{e} \Rightarrow_k e_{\oplus}$ and $\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ \oplus\ \overline{e} \mapsto e_{\oplus}$ so

$$e_{\oplus} \approx_{\mathcal{C}} \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ \oplus\ \overline{e}.$$

case (pv-app)

$$\frac{\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p'\ \overline{e} \Rightarrow_k e}{\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p'c\ \overline{e} \Rightarrow_k e\ [\rho(c)]\ (\mathit{typerec}[\kappa'][\Delta, \eta, \rho]\ c\overline{e})}$$

By induction, $e \approx_{\mathcal{C}} \mathit{typerec}[\Delta, \eta, \rho]\ p'\ \overline{e}$. By definition,

$$\begin{aligned} &e\ [\rho(c)]\ (\mathit{typerec}[\Delta, \eta, \rho]\ c\ \overline{e}) \\ &\quad \approx_{\mathcal{C}} (\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p'\ \overline{e})\ [\rho(c)]\ (\mathit{typerec}[\Delta, \eta, \rho]\ c\ \overline{e}). \end{aligned}$$

As

$$\begin{aligned} &(\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ (p'c)\ \overline{e}) \\ &\quad \mapsto_h (\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ p'\ \overline{e})\ [\rho(c)]\ (\mathit{typerec}[\Delta, \eta, \rho]\ c\ \overline{e}) \end{aligned}$$

they are equivalent under $\approx_{\mathcal{C}}$. By transitivity, we have the desired equivalence.

$\square$

Using the previous lemma about path evaluation, we may show that when a *typerec* term evaluates using the rules of LK, then the resulting term is equivalent to the *typerec* term with respect to the rules of LH.

Lemma 7.1.4 *For all $\Delta \vdash c : \kappa$, and $\Delta'; \Gamma \vdash \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ c\ \overline{e} : [c']\langle \rho(c) : \kappa \rangle$*

$$\textit{if } \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ c\ \overline{e} \mapsto_k e \textit{ then } e \approx_{\mathcal{C}} \mathit{typerec}[\Delta, \eta, \rho]\ c\ \overline{e}.$$

Proof

case $(\kappa \equiv \star)$ Say c weak head normalizes to p, and $\mathit{typerec}[\star][\Delta, \eta, \rho]\ p\ \overline{e} \Rightarrow_k e$. Then $\mathit{typerec}[\star][\Delta, \eta, \rho]\ c\ \overline{e} \mapsto_k e$. By the previous lemma,

$$e \approx_{\mathcal{C}} \mathit{typerec}[\star][\Delta, \eta, \rho]\ p\ \overline{e}.$$

As $\Delta \vdash c = p : \star$, by soundness

$$\mathit{typerec}[\star][\Delta, \eta, \rho]\ c\ \overline{e} \approx_{\mathcal{C}} \mathit{typerec}[\star][\Delta, \eta, \rho]\ p\ \overline{e}.$$

By transitivity

$$e \approx_{\mathcal{C}} \mathit{typerec}[\star][\Delta, \eta, \rho]\ c\ \overline{e},$$

case $(\kappa \equiv \kappa_1 \to \kappa_2)$ Say α is not free in c. As $\Delta \vdash c = \lambda\alpha{:}\kappa_1.c\alpha : \kappa_1 \to \kappa_2$, then

$$\mathit{typerec}[\Delta, \eta, \rho]\ c\ \overline{e} \approx_{\mathcal{C}} \mathit{typerec}[\Delta, \eta, \rho]\ (\lambda\alpha{:}\kappa_1.c\alpha)\ \overline{e}$$

Since both

$$\begin{array}{l} \mathit{typerec}[\kappa_1 \to \kappa_2][\Delta, \eta, \rho]\ c\ \overline{e} \mapsto_k \\ \quad \Lambda\beta{:}\kappa.\lambda x{:}[\gamma.\sigma]\langle \beta : \kappa \rangle.\ \mathit{typerec}[\Delta, \alpha{:}\kappa, \eta, \alpha{:}x, \rho, \alpha{:}\beta]\ (c\alpha)\ \overline{e} \end{array}$$

and

$$\begin{array}{l} \mathit{typerec}[\Delta, \eta, \rho]\ (\lambda\alpha{:}\kappa.c\alpha)\ \overline{e} \mapsto_h \\ \quad \Lambda\beta{:}\kappa.\lambda x{:}[\gamma.\sigma]\langle \beta : \kappa \rangle.\ \mathit{typerec}[\Delta, \alpha{:}\kappa, \eta, \alpha{:}x, \rho, \alpha{:}\beta]\ (c\alpha)\ \overline{e} \end{array}$$

the result follows.

□

Theorem 7.1.5 (Dynamic Correctness) *If $\emptyset \vdash e : \mathit{int}$ then if $e \mapsto_k^* v$ then $e \mapsto_h^* v$.*

Proof

Proof is by induction on n, the number of steps in $e \mapsto_k^n v$. If n is zero then the result follows trivially. Say $e \mapsto_k e'$ and $e' \mapsto_k^n v$. By induction $e' \mapsto_h^* v$. If e is not a *typerec* term the result follows trivially. Say e is $\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ c\ \overline{e}$. By the previous lemma, $e' \approx_{\mathcal{C}} \mathit{typerec}[\kappa][\Delta, \eta, \rho]\ c\ \overline{e}$. By definition of $\approx_{\mathcal{C}}$, as $e' \mapsto_h^* v$ then $\mathit{typerec}[\kappa][\Delta, \eta, \rho]\ c\ \overline{e} \mapsto_h^* v$. □

Table 7.4: LKR: Syntax

$$
\begin{array}{llll}
(kinds) & \kappa & ::= & \star \mid \kappa_1 \to \kappa_1 \\
(cons) & c & ::= & \oplus \mid \alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid R \\
(types) & \sigma & ::= & T(c) \mid int \mid \sigma_1 \to \sigma_2 \mid \forall\alpha{:}\kappa.\sigma \mid R c_1 c_2 \mid \ldots \\
(exps) & e & ::= & i \mid x \mid \lambda x{:}c.e \mid e_1 e_2 \mid fix\ f{:}\sigma.e \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \ldots \\
 & & \mid & R_\oplus \mid typerec[\kappa][c]\ e\ \overline{e} \mid untyrec[\kappa][c]\ e\ \overline{e} \\
(paths) & p & ::= & R_\oplus[c] \mid untyrec[\kappa][c']\ e\ \overline{e} \mid p\ [c]\ e_1 e_2
\end{array}
$$

Table 7.5: Translation of LK to LKR

Type translation

$$
\begin{array}{rcl}
|T(c)| & = & T(c) \\
|\,int\,| & = & int \\
|\sigma_1 \to \sigma_2| & = & |\sigma_1| \to |\sigma_2| \\
|\forall\alpha{:}\kappa.\sigma| & = & \forall\alpha{:}\kappa.\widehat{R}\langle\alpha : \kappa\rangle \to |\sigma|
\end{array}
$$

Term translation

$$
\begin{array}{rcl}
|i| & = & i \\
|\lambda x{:}\sigma.e| & = & \lambda x{:}|\sigma|.|e| \\
|\,fix\ f{:}\sigma.v| & = & fix\ f{:}|\sigma|.|v| \\
|e_1 e_2| & = & |e_1||e_2| \\
|\Lambda\alpha{:}\kappa.e| & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.|e| \\
|e[c]| & = & |e|\ [c]\ \widehat{\mathcal{R}}|c| \\
|\,typerec[\kappa][\Delta,\rho,\eta][c']\ c\ \overline{e}| & = & typerec[\kappa][c']\ \mathcal{R}|c|_{(\emptyset,c',(\Delta,\rho,|\eta|,|\overline{e}|))}\ |\overline{e}|
\end{array}
$$

7.2 Phase-splitting LK

I next present a type-erasure version of higher-order type analysis, called LKR, which uses terms to represent the LK type language. The semantics of this language is mostly determined by a phase-splitting translation of LK, so for presentation I develop it by deriving it from that translation. For reference, the syntax, static semantics and dynamic semantics of LKR language are listed in Tables 7.4, 7.7, and 7.8.

We make one small modification to LK in order to support the phase-splitting translation. Instead of using a type σ with free variable α to describe the return type of a *typerec* expression, we replace this annotation with a type constructor

c' of kind $\star \to \star$. The reason for this restriction is discussed below. Polykinded types indexed by c' (called the *return constructor*) are defined in much the same way as with the $[\alpha.\sigma]$ annotation:

$$[c']\langle c : \star\rangle = T(c'c)$$
$$[c']\langle c : \kappa_1 \to \kappa_2\rangle = \forall\alpha{:}\kappa_1.[c']\langle\alpha : \kappa_1\rangle \to [c']\langle c\alpha : \kappa_2\rangle$$

The phase-splitting translation appears in Table 7.5. It is very similar to the mapping in Chapter 3 from LI to LIR. As before, the goal of the translation is to replace the argument c to *typerec* with its term representation. To create this representation, we must have available the representations of all free type variables that may appear in c'. Therefore, whenever any type variable α is abstracted, its term representation x_α is also abstracted. Whenever a type is instantiated, its term representation is also supplied.

In this translation, we require the following three auxiliary definitions, discussed in the rest of this section. If c is a type argument to a polymorphic term, we must be able to create its representation, which we notate $\widehat{\mathcal{R}}|c|$. We must also define the type of this representation, notated $\widehat{R}\langle c : \kappa\rangle$ where κ is the kind of c. Otherwise, if c is analyzed by *typerec*, we define its representation as $\mathcal{R}|c|_{(\emptyset,c',(\Delta,\rho,|\eta|,|\overline{e}|))}$, parameterized by the components of the *typerec*.

Why does the erasure version of *typerec* not have any environments for the free variables in the constructor c? In LK, the environment η records that a variable α should be interpreted by some term e_α. Here, a term variable (call it y_α) stands for α in the representation of c. Instead of storing in some separate structure a mapping from y_α to e_α, in LKR, we substitute e_α for y_α in the representation. That way, when we evaluate e to a path, we do not need to evaluate it in the presence of free variables (such as y_α).

However, as we analyze the representation of c, we must remember where y_α was so that we may directly return e_α, instead of trying to interpret it as well. Therefore, we need to wrap e_α in a *place holder*—a special term whose only purpose is to mark the presence of e_α. In this calculus, we include a new construct, called *untyrec*, for this purpose. When *typerec* reaches this place holder, e_α is returned.

$$\mathit{typerec}[\kappa][c']\ (\mathit{untyrec}[\kappa][c']\ e_\alpha\ \overline{e})\ \overline{e} \mapsto e_\alpha$$

We also can replace the type environment ρ by substitution. In LK, the type constructor language serves two purposes: describing terms and indexing *typerec*. Consequently, we were forced to delay type instantiations so that they would not interfere with the operation of *typerec*. However, in LKR, term representations are used by *typerec*, so we are free to eagerly substitute these type instantiation.

The difference between using the environments in LK and using substitution (and the place holder) with LKR is apparent in the rule for analyzing constructors of higher kind. In LK, we interpret a constructor function as a term function. In the body of this function, we analyze this constructor by applying it to a fresh variable β. The environment, η, of this analysis is extended so that β will be interpreted as the argument to the function y. Furthermore, if $c\beta$ is ever used as a type we use ρ to replace β with α.

$$\begin{array}{l} \mathit{typerec}[\kappa_1 \to \kappa_2][c'][\Delta, \eta, \rho]\ c\ \overline{e} \mapsto_{\kappa} \\ \quad \Lambda\alpha{:}\kappa.\lambda y{:}[c]\langle\alpha : \kappa\rangle. \\ \qquad \mathit{typerec}[\kappa_2][c'][\Delta, \beta{:}\kappa, \eta, \beta{:}y, \rho, \beta{:}\alpha]\ (c\beta)\ \overline{e} \end{array}$$

In the LKR version of this rule below, assume that e is the representation of c. The operation of this rule also produces a polymorphic function. Because of phase-splitting, this function abstracts x_α, the representation of α, as well as y. The representation e expects the type α (and its representation x_α) as well as its interpretation ($\mathit{untyrec}[\kappa_1][c]\ y\ \overline{e}$).

$$\begin{array}{l} \mathit{typerec}[\kappa_1 \to \kappa_2][c']\ e\ \overline{e} \mapsto_{HR} \\ \quad \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.\lambda y{:}|[c']\langle\alpha : \kappa\rangle|. \\ \qquad \mathit{typerec}[\kappa_2][c']\ \boxed{(e\ [\alpha]\ x_\alpha\ (\mathit{untyrec}[\kappa_1][c']\ y\ \overline{e}))}\ \overline{e} \end{array}$$

7.2.1 A parameterized representation type

For type soundness, we must restrict what terms can be the arguments to *untyrec*—if an arbitrary term were allowed it is not guaranteed that an analysis of a *untyrec* term would result in a term of the correct type. Essentially, *untyrec* coerces any term into a type representation. This coercion is sound if we record what analysis we are allowed to do of this representation. Therefore the LKR version of the R type must be parameterized with an extra argument to describe the result of type analysis allowed for that representation (see Table 7.7). Because this extra argument is a type constructor (of kind $\star \to \star$) we may abstract it. (It is for this reason that we have changed the result annotation of *typerec* from $[\alpha.\sigma]$ to $[c]$). When a term representation is polymorphic over this result constructor, for example, if it is of type $\forall\beta{:}\star \to \star.R\beta c$, then the term may be used for *any* analysis. The following notation stands for this polymorphic type lifted to higher constructors:

Definition 7.2.1 $\widehat{R}\langle c : \kappa\rangle \stackrel{\text{def}}{=} \forall\beta{:}\star \to \star.|[R\beta]\langle c : \kappa\rangle|$

7.2.2 Defining term representations of type constructors

Table 7.6: Representation of constructor language

$$
\begin{array}{rcl}
\Psi & = & (\Delta, \rho, \eta, \overline{e}) \mid \bullet \\
\Theta & = & (\Delta', c', \Psi) \\
\widehat{\mathcal{R}}|c| & = & \Lambda\alpha{:}\star \to \star.\mathcal{R}|c|_{(\emptyset,\alpha,\bullet)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{R}|\oplus|_\Theta & = & R_\oplus[c'] \\
\mathcal{R}|\alpha|_\Theta & = & \begin{cases} \mathit{untyrec}[\Delta(\alpha)][c']\ \eta(\alpha)\ \overline{e} & \text{if } \Psi \text{ is not } \bullet \text{ and } \alpha \in \mathit{Dom}\,\Delta \\ y_\alpha & \text{if } \alpha \in \mathit{Dom}\,\Delta' \\ x_\alpha[c'] & \text{otherwise} \end{cases} \\
\mathcal{R}|\lambda\alpha{:}\kappa.c_1|_\Theta & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.\lambda y_\alpha{:}|[Rc']\langle\alpha : \kappa\rangle|.\mathcal{R}|c_1|_{(\Delta',\alpha:\kappa,c',\Psi)} \\
\mathcal{R}|c_1 c_2|_\Theta & = & \mathcal{R}|c_1|_\Theta\ [\rho(c_2)]\ \widehat{\mathcal{R}}|\rho(c_2)|\ \mathcal{R}|c_2|_\Theta
\end{array}
$$

There are two sorts of term representations in this calculus. The first sort, called *open representations*, represent types that are in the process of being analyzed, i.e. terms that represent arguments to *typerec*: $\mathcal{R}|c|_{(\emptyset,c',(\Delta,\rho,|\eta|,|\overline{e}|))}$ (described below). They are called open because they may contain free variables for which the LK *typerec* provides an interpretation. These representations are of type $|[Rc']\langle c : \kappa\rangle|$, where the index c' indicates the result type of analysis. Such representations may be used *only* in an analysis that produces a result of type $|[c']\langle c : \kappa\rangle|$. The reason is because, inside these representations, there may be *untyrec* terms holding the results of analysis of the free variables—and those results must agree with c'.

The second sort of term representations are the *closed representations*, $\widehat{\mathcal{R}}|c|$, of type $\widehat{R}\langle c : \kappa\rangle = \forall\beta{:}\star \to \star.|[R\beta]\langle c : \kappa\rangle|$. These representations are not arguments to *typerec*—they are polymorphic with respect to the result constructor of any possible analysis. That polymorphism must be instantiated to the appropriate type constructor before they may be analyzed. Because *untyrec* terms depend on this instantiation, term representations of this sort cannot have any *untyrec* expressions as subterms, and are hence closed.[2]

[2] Here the polymorphism of the return constructor acts like a closure operator and is similar to the modal box types of Schürmann, Pfenning, and Despeyroux[SDP01],[DL01]. These types play a role in induction over higher-order abstract syntax.

We may construct the closed representation as a special case of the open representation, defined as $\mathcal{R}|c|_\Theta$ in Table 7.6. The parameter Θ describes the context of this representation (Δ'), the return constructor (c'), and whether this is an open or closed representation Ψ. If this is an open representation, $\Psi = (\Delta, \rho, \eta, \overline{e})$, holding the context, environments and branches from an enclosing context. In that event, the translation may construct an appropriate *untyrec* placeholder for variables in Δ. Otherwise, for a closed representation $\Psi = \bullet$.

The tricky part of this translation is the case for variables. If the constructor variable is in Δ (if Ψ is not $\bullet$) then the variable was bound by an enclosing *typerec*, and there is a binding for it within η. This result should be wrapped by an *untyrec* so that analysis should produce the correct result. Otherwise, if the variable is in Δ', then this variable is bound by some type-level λ, and there will be a closed representative y_α. As this representative is specialized to c', we can immediately return it. Otherwise, this variable was bound by some term-level Λ, and there is some associated x_α that represents it. However this representative is polymorphic over the return constructor, so we need to instantiate it with c'.

The representations of type-level abstractions are polymorphic functions, abstracting both the closed representations x_α and the open representations y_α. Likewise, the representations of type-level applications provide both the closed and open representations of the type argument, c_2. Note that we use ρ to substitute for any type variables in c_2.

For example, the closed representation of $\lambda\alpha{:}\star.\alpha \times int$ is below.

$$\begin{array}{l}\Lambda\beta{:}\star \to \star.\Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \star\rangle.\lambda y_\alpha{:}|[\beta]\langle\alpha : \star\rangle|. \\ \quad R_\times[\beta]\ [\alpha]\ x_\alpha\ y_\alpha\ [int]\ \ R_{int}\ (R_{int}[\beta])\end{array}$$

In this representation, we instantiate $R_\times$ with the return constructor β, the first component of the product type α, along with its open representation $x_\alpha$and its closed representation y_α, and the second component of the product type *int*, along with its open representation R_{int} and its closed representation $R_{int}[\beta]$.

7.3 The LKR language

7.3.1 Static semantics

Table 7.7 shows the static semantics for the representation terms, *typerec* and *untyrec*. If $\oplus$ is an arbitrary type constructor constant, such as *int*, $\times$, or $\to$, in LKR, $R_\oplus$ is its term representation. If $\oplus$ is of kind $\kappa_\oplus$, then the type of $R_\oplus$ is $\widehat{R}\langle\oplus : \kappa_\oplus\rangle$.

Table 7.7: LKR: Static semantics

$\boxed{\Delta \vdash c : \kappa}$

[*c-R*]

$$\frac{}{\Delta \vdash R \ : (\star \to \star) \to \star \to \star}$$

$\boxed{\Delta \vdash \sigma = \sigma'}$

[*ceq-R*]

$$\frac{\Delta \vdash c : \star \to \star \qquad \Delta \vdash \tau : \star}{\Delta \vdash T(R\ c\ \tau) = R\ c\ \tau}$$

[*ceq-alltype*]

$$\frac{\Delta \vdash c : \star \to \star}{\Delta \vdash T(\forall_\star\ c) = \forall \alpha{:}\star . \widehat{R}\langle \alpha : \kappa \rangle \to T(c\alpha)}$$

$\boxed{\Delta \vdash \sigma}$

[*t-R*]

$$\frac{\Delta \vdash c : \star \to \star \qquad \Delta \vdash \tau : \star}{\Delta \vdash R\ c\ \tau}$$

$\boxed{\Delta; \Gamma \vdash e : \sigma}$

[*e-tyrep*]

$$\frac{}{\Delta; \Gamma \vdash R_\oplus : \widehat{R}\langle \oplus : \kappa_\oplus \rangle}$$

[*e-typerec*]

$$\frac{\begin{array}{l} \Delta \vdash c : \kappa \\ \Delta \vdash c' : \star \to \star \\ \Delta; \Gamma \vdash e_\oplus : |[c']\langle \oplus : \kappa_\oplus \rangle| \qquad (e_\oplus \in \overline{e}) \\ \Delta; \Gamma \vdash e : |[Rc']\langle c : \kappa \rangle| \end{array}}{\Delta; \Gamma \vdash \mathit{typerec}[\kappa][c']\ e\ \overline{e} : |[c']\langle c : \kappa \rangle|}$$

[*e-untyrec*]

$$\frac{\begin{array}{l} \Delta \vdash c : \kappa \\ \Delta \vdash c' : \star \to \star \\ \Delta; \Gamma \vdash e_\oplus : |[c']\langle \oplus : \kappa_\oplus \rangle| \qquad (e_\oplus \in \overline{e}) \\ \Delta; \Gamma \vdash e : |[c']\langle c : \kappa \rangle| \end{array}}{\Delta; \Gamma \vdash \mathit{untyrec}[\kappa][c']\ e\ \overline{e} : |[Rc']\langle c : \kappa \rangle|}$$

Table 7.8: LKR: Operational semantics for *typerec*

$\boxed{e \Rightarrow_{HR} e'}$

[*pv-var*]
$$\frac{}{typerec[\kappa][c']\ (untyrec[\kappa][c']\ e\ \overline{e})\ \overline{e}' \Rightarrow_{HR} e}$$

[*pv-const*]
$$\frac{}{typerec[\kappa][c']\ (R_{\oplus}[c'])\ \overline{e}\ \ \Rightarrow_{HR} e_{\oplus}}$$

[*pv-app*]
$$\frac{typerec[\kappa_1 \rightarrow \kappa][c']\ p\ \overline{e} \Rightarrow_{HR} e'}{typerec[\kappa][c']\ (p\ [c]\ e_1\ e_2)\ \overline{e} \Rightarrow_{HR} e'\ [c]\ e_1\ (typerec[\kappa_1][c']\ e_2\ \overline{e})}$$

$\boxed{e \mapsto_{HR} e'}$

[*ev-trec-path*]
$$\frac{typerec[\star][c']\ p\ \overline{e} \Rightarrow_{HR} e'}{typerec[\star][c']\ p\ \overline{e} \mapsto_{HR} e'}$$

[*ev-trec-cong*]
$$\frac{e \mapsto e'}{typerec[\star][c']\ e\ \overline{e} \mapsto_{HR} typerec[\star][c']\ e'\ \overline{e}}$$

[*ev-trec-arrow*]
$$\frac{}{\begin{array}{l} typerec[\kappa_1 \rightarrow \kappa_2][c']\ e\ \overline{e} \mapsto_{HR} \\ \quad \Lambda\alpha{:}\kappa_1.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa_1\rangle.\lambda y{:}|[c']\langle\alpha : \kappa_1\rangle|. \\ \quad\quad typerec[\kappa_2][c']\ (e\ [\alpha]\ (\Lambda\beta{:}\star \rightarrow \star.x_\alpha[\beta])\ (untyrec[\kappa_1][c']\ y\ \overline{e}))\ \overline{e} \end{array}}$$

The last two formation rules in the table are for *typerec* and *untyrec*. In these rules, the branches are of the same types as in the previous languages. However, the argument to *typerec* must be a term representation, and the result of *untyrec* is also a term representation. From the typing judgment, observe these two terms are inverses of each other, coercing between $|[Rc']\langle c : \kappa\rangle|$ and $|[c']\langle c : \kappa\rangle|$.

7.3.2 Dynamic semantics

Table 7.8 presents the dynamic semantics of the erasure language. Again reduction is divided between reduction of paths p (notated by $\Rightarrow$) and the kind-directed small step reduction (notated by $\mapsto$). A path in this language is the term representation of a path in LK. It is the representation of an operator, the representation of a variable with *untyrec*, or the representation of a path application, where e_1 is the closed representation of path argument c, and e_2 is the open representation.

The rules for path evaluation and the small-step semantics for *typerec* are a translation of the rules of LK. For example as $R_\oplus[c']$ is the translation of the path $\oplus$, as in LK, path evaluation produces $e_\oplus$. The only exception is for *typerec* when the argument is of higher-kind. Instead of applying the representation e to x_α as we discussed earlier, we η-expand that variable to $\Lambda\beta{:}\star \to \star . x_\alpha[\beta]$. This small change simplifies the proof of dynamic correctness.

7.4 An example

As an example of a function written in this language, Figure 7.1 contains the function *copy* from the previous chapter. Like before, this function analyzes its type argument α to return a copying function for objects of type α.

This function demonstrates the differences between LK and LKR. First, all type abstractions in this version of *copy* are immediately followed by an abstraction of a term representation. For example, not only does *copy* abstract α, it also abstracts its representation x_α. It is this representation x_α that is the argument to *typerec*. Because x_α is polymorphic over the return type of analysis (it is of type $\widehat{R}\langle\alpha : \star\rangle = \forall\gamma{:}\star \to \star . R\gamma\alpha$), it must be instantiated with $\lambda\beta{:}\star . \beta \to \beta$ before it may be analyzed.

Compare the $\mu_\star$ branch in the LKR version with that of LK below.

$$\begin{aligned}
\mu_\star \Rightarrow\ & \Lambda\alpha{:}\star \to \star . \\
& \quad \lambda r{:}(\forall\beta{:}\star . T(\beta \to \beta) \to T(\alpha\beta \to \alpha\beta)). \\
& \quad\ \ \mathit{fix}\ f{:}T(\mu_\star\alpha \to \mu_\star\alpha).\lambda x{:}T(\mu_\star\alpha). \\
& \quad\quad \mathit{roll}\ (r\ [\mu_\star\alpha]\ f\ (\mathit{unroll}\,x))
\end{aligned}$$

In the LKR version, we need to abstract the representation of α. Furthermore, because r quantifies over the type β, it also requires the representation of β. So when r is instantiated with $\mu_\star\alpha$, it must also be supplied with the representation of $\mu_\star\alpha$ as well.

What is the representation of $\mu_\star\alpha$? It is $(\Lambda\beta . R_{\mu_\star}[\beta][\alpha]\ x_\alpha\ (x_\alpha[\beta]))$. This representation must be applicable for any iteration, so it must abstract the return type constructor β. The representation of $\mu_\star\alpha$ is the representation of $\mu_\star$, $R_{\mu_\star}[\beta]$, applied to the closed representation of α, which is x_α, then applied to the open representation of α in this context, which is $x_\alpha[\beta]$.

$$
\begin{array}{l}
\textit{fix copy} : (\forall\alpha{:}\star.\widehat{R}\langle\alpha:\star\rangle \to T(\alpha\to\alpha)).\\
\quad \Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\rangle.\\
\quad\quad \textit{typerec}[\star][\lambda\beta.\beta\to\beta]\ (x_\alpha[\lambda\beta.\beta\to\beta])\ \ \textit{of}\\
\quad\quad\ \textit{int} \quad \Rightarrow \lambda i{:}\textit{int}.i\\
\quad\quad\ \to \quad \Rightarrow \Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\rangle.\lambda r_\alpha{:}T(\alpha\to\alpha).\\
\quad\quad\quad\quad\quad \Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta:\star\rangle.\lambda r_\beta{:}T(\beta\to\beta).\\
\quad\quad\quad\quad\quad\quad \lambda f{:}T(\alpha\to\beta).r_\beta\circ f\circ r_\alpha\\
\quad\quad\ \times \quad \Rightarrow \Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\rangle.\lambda r_\alpha{:}T(\alpha\to\alpha).\\
\quad\quad\quad\quad\quad \Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta:\star\rangle.\lambda r_\beta{:}T(\beta\to\beta).\\
\quad\quad\quad\quad\quad\quad \lambda x{:}T(\alpha\times\beta).\langle r_\alpha(\pi_1 x), r_\beta(\pi_2 x)\rangle\\
\quad\quad\ \mu_\star \quad \Rightarrow \Lambda\alpha{:}\star\to\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\to\star\rangle.\\
\quad\quad\quad\quad\quad\quad \lambda r{:}(\forall\beta{:}\star.\widehat{R}\langle\beta:\star\rangle\to T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
\quad\quad\quad\quad\quad\quad\quad \textit{fix } f{:}T(\mu_\star\alpha\to\mu_\star\alpha).\lambda x{:}T(\mu_\star\alpha).\\
\quad\quad\quad\quad\quad\quad\quad\quad \textit{roll}\ (r\ [\mu_\star\alpha]\ (\Lambda\beta.R_{\mu_\star}[\beta][\alpha]\ x_\alpha\ (x_\alpha[\beta]))\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad f\ (\textit{unroll}\,x))\\
\quad\quad\ \forall_\star \quad \Rightarrow \Lambda\alpha{:}\star\to\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\to\star\rangle.\\
\quad\quad\quad\quad\quad\quad \lambda r{:}(\forall\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta:\star\rangle.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
\quad\quad\quad\quad\quad\quad\quad \lambda x{:}T(\forall_\star\alpha).\\
\quad\quad\quad\quad\quad\quad\quad\quad \Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta:\star\rangle.r[\beta]\ x_\beta\ (\textit{copy}[\beta]\ x_\beta)(x\ [\beta]\ x_\beta)\\
\quad\quad\ \exists_\star \quad \Rightarrow \Lambda\alpha{:}\star\to\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\to\star\rangle.\\
\quad\quad\quad\quad\quad\quad \lambda r{:}(\forall\beta{:}\star.\widehat{R}\langle\beta:\star\rangle\to T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
\quad\quad\quad\quad\quad\quad\quad \lambda x{:}T(\exists_\star\alpha).\\
\quad\quad\quad\quad\quad\quad\quad\quad \textit{let}\langle\beta,\langle x_\beta,y\rangle\rangle = \textit{unpack}\,x\,\textit{in}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textit{pack}\langle\beta,\langle x_\beta, r[\beta]\ x_\beta\ (\textit{copy}[\beta]\ x_\beta)\ y\rangle\rangle\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textit{as}\ \exists\beta{:}\star.\widehat{R}\langle\beta:\star\rangle\times\alpha\beta
\end{array}
$$

Figure 7.1: Example: Erasure version of *copy*

Now compare the $\forall_\star$ branch in the LKR version with that of LK below:

$$
\begin{array}{l}
\forall_\star \Rightarrow \Lambda\alpha{:}\star\to\star.\\
\quad\quad \lambda r{:}(\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
\quad\quad\quad \lambda x{:}T(\forall_\star\alpha).\\
\quad\quad\quad\quad \Lambda\beta{:}\star.r\ [\beta]\ (\textit{copy}[\beta])(x[\beta])
\end{array}
$$

Again r requires the representation of β. This example shows the difference between the interpretation of $\forall_\star$ in LK and in LKR. In LK

$$T(\forall_\star \alpha) = \forall \beta{:}\star\,.\,T(\alpha\beta)$$

while in LKR, the interpretation of the $\forall_\star$ constructor must take into account the type translation (see Table 7.7):

$$T(\forall_\star \alpha) = \forall \beta{:}\star\,.\widehat{R}\langle \beta : \star\rangle \rightarrow T(\alpha\beta).$$

This change in interpretation is necessary in order to *copy* in LKR. In the calls to r and to *copy* in this branch, we must supply not just the type β, but its representation as well. With this rule, the resulting function in this branch must be of type

$$(\forall \beta{:}\star\,.\widehat{R}\langle \beta : \star\rangle \rightarrow T(\alpha\beta)) \rightarrow (\forall \beta{:}\star\,.\widehat{R}\langle \beta : \star\rangle \rightarrow T(\alpha\beta)).$$

Therefore we may abstract x_β in the last line of the branch, and use it as the needed arguments.

7.5 Typing properties of LKR

In order to prove type safety of this language we will need to show the usual subject reduction and progress lemmas.

As before, type transformation commutes with substitution.

Lemma 7.5.1 $|\sigma|[c/\alpha] = |\sigma[c/\alpha]|$

The substitution only occurs inside constructors buried in the type σ below. These constructors are unchanged by type transformation.

As is standard, LKR possesses the same substitution properties as the previous languages.

Lemma 7.5.2 (Substitution) *1. If* $\Delta, \alpha{:}\kappa' \vdash c : \kappa$ *and* $\Delta \vdash c' : \kappa'$ *then* $\Delta \vdash c[c'/\alpha] : \kappa$.

2. If $\Delta, \alpha{:}\kappa' \vdash c_1 = c_2 : \kappa$ *and* $\Delta \vdash c' : \kappa'$ *then* $\Delta[c'/\alpha] \vdash c_1[c'/\alpha] = c_2[c'/\alpha] : \kappa$.

3. If $\Delta, \alpha{:}\kappa \vdash \sigma$ *and* $\Delta \vdash c : \kappa$ *then* $\Delta[c/\alpha] \vdash \sigma[c/\alpha]$.

4. If $\Delta, \alpha{:}\kappa \vdash \sigma = \sigma'$ *and* $\Delta \vdash c : \kappa$ *then* $\Delta[c/\alpha] \vdash \sigma[c/\alpha] = \sigma'[c/\alpha]$.

5. If $\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$ *and* $\emptyset; \emptyset \vdash c : \kappa$ *then* $\Delta; \Gamma[c/\alpha] \vdash e[c/\alpha] : \sigma[c/\alpha]$.

6. *If* $\Delta; \Gamma, x{:}\sigma' \vdash e : \sigma$ *and* $\emptyset; \emptyset \vdash e' : \sigma'$ *then* $\Delta; \Gamma \vdash e[e'/x] : \sigma$*.*

Proof

The proofs these substitution lemmas are more straightforward than that of LH because *typerec* (and *untyrec*) do not bind any type or term variables. Therefore, I will only give the case for *untyrec* for the proof of constructor substitution in term judgments.

case e-*typerec*. Suppose

$$\frac{\begin{array}{ll} \Delta, \alpha{:}\kappa' \vdash c_1 : \kappa & \\ \Delta, \alpha{:}\kappa' \vdash c' : \star \to \star & \\ \Delta, \alpha{:}\kappa'; \Gamma \vdash e_\oplus : |[c']\langle \oplus : \kappa_\oplus \rangle| & (e_\oplus \in \overline{e}) \\ \Delta, \alpha{:}\kappa'; \Gamma \vdash e : |[c']\langle c_1 : \kappa \rangle| & \end{array}}{\Delta, \alpha{:}\kappa'; \Gamma \vdash \mathit{untyrec}[\kappa][c']\ e\ \overline{e} : |[Rc']\langle c_1 : \kappa \rangle|}$$

By induction

$$\begin{array}{ll} \Delta \vdash c_1[c/\alpha] : \kappa & \\ \Delta \vdash c'[c/\alpha] : \star \to \star & \\ \Delta; \Gamma[c/\alpha] \vdash e_\oplus[c/\alpha] : |[c']\langle \oplus : \kappa_\oplus \rangle|[c/\alpha] & (e_\oplus \in \overline{e}) \\ \Delta; \Gamma[c/\alpha] \vdash e[c/\alpha] : |[c']\langle c_1 : \kappa \rangle|[c/\alpha] & \end{array}$$

Therefore $\Delta; \Gamma[c/\alpha] \vdash (\mathit{untyrec}[\kappa][c']\ e\ \overline{e})[c/\alpha] : |[c']\langle c_1 : \kappa \rangle|[c/\alpha]$.

□

Lemma 7.5.3 (Subject Reduction for paths) *If*

$$\emptyset; \emptyset \vdash \mathit{typerec}[\kappa][c']\ e\ \overline{e} : [c']\langle c : \kappa \rangle$$

and if e *is a path and* $\mathit{typerec}[\kappa][c']\ e\ \overline{e} \Rightarrow_{HR} e'$ *then* $\emptyset; \emptyset \vdash e' : [c']\langle c : \kappa \rangle$*.*

Proof

By induction on $\mathit{typerec}[\kappa][c']\ e\ \overline{e} \Rightarrow_{HR} e'$.

case pv-var Here, e is $(\mathit{untyrec}[\kappa][c']\ e'\ \overline{e}')$ and path evaluation steps to e'. As the left hand side was well typed, then by inversion $\emptyset; \emptyset\ \mathit{untyrec}[\kappa][c']\ e'\ \overline{e}' : |[Rc']\langle c : \kappa \rangle|$. Again by inversion $\emptyset; \emptyset e : |[c']\langle c : \kappa \rangle|$, the type of the left hand side.

case pv-const In this case, e is $R_{int}[c']$ and path evaluation steps to $e_{\oplus}$. As $R_{int}[c'] : |[c']\langle \oplus : \kappa_{\oplus}\rangle|$, then $\emptyset; \emptyset \vdash \mathit{typerec}[\kappa_{\oplus}][c']\ \ R_{int}[c']\ \overline{e} : |[c']\langle \oplus : \kappa_{\oplus}\rangle|$. By inversion of this judgment, we derive that the result has the same type: $\emptyset; \emptyset \vdash e_{\oplus} : |[c']\langle \oplus : \kappa_{\oplus}\rangle|$.

case pv-app Finally e is $p\ [c_2]\ e_1\ e_2$ and path evaluation steps to

$$e'\ [c]\ e_1\ (\mathit{typerec}[k_1][c']\ e_2\ \overline{e})$$

when $\mathit{typerec}[\kappa_1 \to \kappa][c']\ p\ \overline{e} \Rightarrow_{HR} e'$. By inversion,

$$\emptyset; \emptyset \vdash p\ [c_2]\ e_1\ e_2 : |[Rc']\langle c_1 c_2 : \kappa\rangle|$$

and by inversion of this judgment we know:

$$\begin{array}{l}
\emptyset \vdash c_1 : \kappa_1 \to \kappa \\
\emptyset; \emptyset \vdash p : |[Rc']\langle c_1 : \kappa_1 \to \kappa\rangle| \\
\emptyset \vdash c_2 : \kappa_1 \\
\emptyset; \emptyset \vdash \mathit{typerec}[\kappa_1 \to \kappa][c']\ e_1 : \widehat{R}\langle c_2 : \kappa_1\rangle \\
\emptyset; \emptyset \vdash \mathit{typerec}[\kappa_1 \to \kappa][c']\ e_2 : |[Rc']\langle c_2 : \kappa_1\rangle|
\end{array}$$

From the above, we may conclude $\emptyset; \emptyset \vdash \mathit{typerec}[\kappa_1 \to \kappa][c']\ p\ \overline{e} : |[c']\langle c_1 : \kappa_1 \to \kappa\rangle|$ so by induction, e' is also of type $|[c']\langle c_1 : \kappa_1 \to \kappa\rangle|$. By definition, this type is equal to $\forall \alpha{:}\kappa_1.\widehat{R}\langle \alpha : \kappa_1\rangle \to |[c']\langle \alpha : \kappa_1\rangle| \to |[c']\langle c_1 a : |\rangle$. Therefore, after the type and term applications we may show that

$$\emptyset; \emptyset \vdash p\ [c_2]\ e_1\ e_2 : |[c']\langle c_1 c_2 : \kappa\rangle|$$

□

Lemma 7.5.4 (Subject Reduction) *If* $\emptyset; \emptyset \vdash e : \sigma$ *and* $e \mapsto e'$ *then* $\emptyset; \emptyset \vdash e' : \sigma$.

Proof

By induction on $e \mapsto e'$. Below are the cases in which e is a *typerec* term, so $\mathit{typerec}[\kappa][c']\ e\ \overline{e} \mapsto_{HR} e'$.

then

case ev-trec-path This case follows directly by the previous lemma.

case ev-trec-cong This case follows directly by induction.

case ev-trec-arrow In this case, e represents the constructor c of kind $\kappa_1 \to \kappa$, so we wish to show that $\emptyset;\emptyset \vdash e' : |[c']\langle c : \kappa_1 \to \kappa\rangle|$, where e' is

$$\begin{array}{l}\Lambda\alpha{:}\kappa_1.\lambda x_\alpha{:}\widehat{R}\langle \alpha : \kappa_1\rangle.\lambda y{:}|[c']\langle \alpha : \kappa_1\rangle|.\\ \quad \mathit{typerec}[\kappa][c']\ (e\ [\alpha](\Lambda\beta{:}\star \to \star.x_\alpha[\beta])\ (\mathit{untyrec}[\kappa_1][c']\ y\ \overline{e}))\overline{e}\end{array}$$

By inversion

$$\begin{array}{l}\emptyset \vdash c' : \star \to \star\\ \emptyset \vdash c : \kappa_1 \to \kappa \emptyset;\emptyset \vdash e : |[c']\langle c : \kappa_1 \to \kappa\rangle|\\ \emptyset;\emptyset \vdash e_\oplus : |[c']\langle c : \kappa_1 \to \kappa\rangle|\end{array}$$

Let $\Delta';\Gamma' = \Delta, \alpha{:}\kappa_1; \Gamma, x_\alpha{:}\widehat{R}\langle \alpha : \kappa_1\rangle, y{:}|[c']\langle \alpha : \kappa_1\rangle|$. From the above we may show

$$\Delta';\Gamma' \vdash (\mathit{untyrec}[\kappa_1][c']\ y\ \overline{e}) : |[Rc']\langle \alpha : \kappa_1\rangle|$$

as e is of type $\forall\alpha{:}\kappa_1.\widehat{R}\langle \alpha : \kappa_1\rangle \to |[c']\langle \alpha : \kappa_1\rangle| \to |[c']\langle c\alpha : \kappa\rangle|$, we can show

$$\Delta';\Gamma' \vdash e\ [\alpha]\ (\Lambda\beta{:}\star \to \star.x_\alpha[\beta])\ (\mathit{untyrec}[\kappa_1][c']\ y\ \overline{e}) : |[c']\langle c\alpha : \kappa\rangle|$$

Therefore e' has the correct type.

□

Because our calculus is call-by-name, we have not yet defined the value forms. However, we need that definition in order to state the Progress lemma.

Definition 7.5.5 (LKR values)

$$v ::= i \mid \lambda x{:}\sigma.e \mid \mathit{fix}\ f{:}\sigma.e \mid \Lambda\alpha{:}\kappa.e \mid p$$

For progress we must show a special form of the canonical forms lemma:

Lemma 7.5.6 *If $\emptyset \vdash e : R\ c'\ c$ and e is a value then e is a path.*

Proof is by examination of the value forms : integers, term and type abstractions cannot produce a term of type $R\ c'\ c$.

We also need the same canonical forms lemma as in LI:

Lemma 7.5.7 (Canonical forms) *If $\emptyset \vdash v : \sigma$ then*

If $\emptyset \vdash \sigma = \mathit{int}$ then v is i.

If $\emptyset \vdash \sigma = \sigma_1 \to \sigma_2$ then v is either $\lambda x{:}\sigma_1.e$ or $(\mathit{fix}\ f{:}(\sigma_1 \to \sigma_2).v')[c_1]\cdots[c_n]$.

If $\emptyset \vdash \sigma = \forall\alpha{:}\kappa.\sigma_1$ then v is either $\Lambda\alpha{:}\kappa.v'$ or $(\mathit{fix}\ f{:}(\alpha{:}\kappa.\sigma_1).v')[c_1]\cdots[c_n]$.

Lemma 7.5.8 (Progress for LKR) *If $\emptyset \vdash e : \sigma$ then either e is a value or there exists an e' such that $e \mapsto e'$.*

Proof

By induction on $\emptyset \vdash e : \sigma$. Consider the case where $\emptyset \vdash \mathit{typerec}[\kappa][\alpha.\sigma]e'\overline{e} : [Rc']\langle c : \kappa\rangle$. If κ is $\kappa_1 \to \kappa_2$ then [ev-trec-arrow] applies. Otherwise, if e is not a path then it must not be a value so by induction the term steps by [ev-trec-cong]. Otherwise, if e is a path, then one of the three path evaluation rules applies.

□

7.6 Correctness of the embedding of LK

7.6.1 Static correctness

The static correctness of this translation can be shown in a manner similar to that of the translation from LI to LIR, in Chapter 3. Because we essentially have two versions of type representations—one for constructors that may have variables bound by an enclosing *typerec*, and one for constructors that are in other contexts, we must have two lemmas about the type soundness of the translation. Furthermore, we have two translations of Δ to produce the context for the type representations variables. In the first case, the translation is specialized by a return type constructor. The type of each representation variable must be specialized to this constructor. In the second case, for those variables bound by a term-level type abstraction (Λ), the types of the representations must be polymorphic over the return type.

$$|\Delta, \alpha{:}\kappa|_c = |\Delta|_c, y_\alpha : |[c]\langle \alpha : \kappa\rangle|$$
$$|\Delta, \alpha{:}\kappa| = |\Delta|, x_\alpha : \widehat{R}\langle \alpha : \kappa\rangle$$

In the following two lemmas, we show that the representation of a LK constructor c is well-typed. The free variables of c may be bound in many different situations. We let Δ_1 refer to all of those bound by enclosing term-level type abstractions (Λ), Δ_2 refer to variables bound by type level type abstractions (λ), and Δ_3 list variables bound by enclosing *typerec* expressions. There are two versions of this lemma: this first for when the constructor does not appear inside of a *typerec*, and the second, when we must add the Ψ component to the representation. Much of the proof of this lemma is similar to the proof of Lemma 3.4.2, that the representation of a constructor in LIR is of the correct type.

Lemma 7.6.1 *Let $\Delta = \Delta_1, \Delta_2$. If*

1. $\Delta \vdash c : \kappa$

2. $\Delta_1, \Delta_2 \vdash c' : \star \to \star$

then

$$\Delta_1\Delta_2; |\Delta_1|, |\Delta_2|_{c'} \vdash \mathcal{R}|c|_{(\Delta_2,c',\bullet)} : |[Rc']\langle c : \kappa\rangle|$$

Lemma 7.6.2 *Let* $\Delta = \Delta_1, \Delta_2, \Delta_3$ *and* $\Psi = (\Delta_3, |\eta|, \rho, |\overline{e}|)$. *If*

1. $\Delta \vdash c : \kappa$

2. $\Delta_1, \Delta_2 \vdash c' : \star \to \star$

3. $\Delta_1, \Delta_2; \Gamma[c'] \vdash \Delta_3 \mid \eta \mid \rho$

4. $\Delta_1, \Delta_2; \Gamma \vdash e_\oplus : [c]\langle \oplus : \kappa_\oplus\rangle$ *for* $(e_\oplus \in \overline{e})$.

then

$$\Delta_1\Delta_2; |\Delta_1|, |\Delta_2|_{c'} \vdash \mathcal{R}|c|_{(\Delta_2,c',\Psi)} : |[Rc']\langle \rho(c) : \kappa\rangle|$$

Proof

The proofs of both of these lemmas are very similar, by induction on the derivation of $\Delta \vdash c : \kappa$. In fact, the second lemma is a generalization of the first. We state them separately as the second requires the first lemma in the case of constructor applications, but the proofs of both follow the form below.

case (c-var) For the first lemma, there are two cases of variables corresponding to the contexts that could bind them.

- In the first case, α is bound by a Λ abstraction and is in Δ_1. Therefore, $\mathcal{R}|\alpha|_{(\Delta_2,c',\Psi)} = x_\alpha[c']$. The binding for x_α comes from the translation of $|\Delta_1|$: as x_α is of type

$$\forall\beta{:}\star \to \star.|[R\beta]\langle \alpha : \kappa\rangle|,$$

  the representation is of type $|[Rc']\langle \alpha : \kappa\rangle|$.

- Say α is bound by a constructor abstraction and is in Δ_2. Then the representation of α is y_α, which has the appropriate type by the translation $|\Delta_2|_c$.

- In the second lemma, α could additionally be bound by Δ_3, the context in a *typerec* expression. In this case, the representation of α is *untyrec*$[\kappa][c']\ |\eta(\alpha)|\ |\overline{e}|$. Because the environment η is well-formed, as are the branches $\overline{e}$, the type of the term is $|[Rc']\langle \alpha : \kappa\rangle|$.

case (c-const) By definition

$$\frac{\Delta \vdash R_{\oplus} : \forall\beta{:}\star \to \star.|[R\beta]\langle \oplus : \kappa\rangle| \qquad \Delta \vdash c : \star \to \star}{\Delta \vdash R_{\oplus}[c'] : |[Rc']\langle \oplus : \kappa\rangle|}$$

case (c-app) Say the last step of the derivation was

$$\frac{\Delta \vdash c_1 : \kappa' \to \kappa \qquad \Delta \vdash c_2 : k'}{\Delta \vdash c_1 c_2 : \kappa}$$

The proofs of the two lemmas are not exactly the same for this case. For the first lemma, we know by induction that the type of $\widehat{\mathcal{R}}|c_1|$ is

$$|[Rc']\langle c_1 : \kappa' \to \kappa_2\rangle| = \forall\alpha{:}\kappa'.\widehat{R}\langle\alpha : \kappa'\rangle \to |[Rc']\langle\alpha : \kappa'\rangle| \to |[Rc]\langle c_1\alpha : \kappa\rangle|$$

and the type of $\widehat{\mathcal{R}}|c_2|$ is $|[Rc]\langle c_2 : \kappa'\rangle|$. Therefore the type of

$$\widehat{\mathcal{R}}|c_1|\ [c_2]\ \widehat{\mathcal{R}}|c_2|\ \widehat{\mathcal{R}}|c_2|$$

is $[Rc']\langle c_1 c_2 : \kappa\rangle$.

For the second lemma, the case proceed similarly, except that we must show that the type of $\widehat{\mathcal{R}}|\rho(c_2)|$ is $\widehat{R}\langle\rho(c_2) : \kappa'\rangle$. Because $\Delta_1, \Delta_2 \vdash \rho(c_2) : \kappa'$ (by LI constructor substitution), we may conclude this judgment using the first lemma.

Therefore the type of

$$\mathcal{R}|c_1|_{\Theta}\ [\rho(c_2)]\ \widehat{\mathcal{R}}|\rho(c_2)|\ \mathcal{R}|c_2|_{\Theta}$$

is $[Rc']\langle\rho(c_1 c_2) : \kappa\rangle$.

case (c-abs) Say the last step of the derivation was

$$\frac{\Delta_1, \Delta_2, \alpha{:}\kappa', \Delta_3 \vdash c : \kappa}{\Delta_1, \Delta_2, \Delta_3 \vdash \lambda\alpha{:}\kappa'.c : \kappa' \to \kappa}$$

By induction, we may conclude

$$\Delta_1, \Delta_2, \alpha{:}\kappa'; |\Delta_1|, |\Delta_2|_c, y_\alpha{:}|[Rc']\langle\alpha : \kappa'\rangle| \vdash \mathcal{R}|c|_{\Theta} : |[Rc']\langle\rho(c) : \kappa\rangle|$$

With weakening, this leads to

$$\begin{array}{l}\Delta_1, \Delta_2; |\Delta_1|, |\Delta_2|_c \vdash \\ \quad \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.\lambda y_\alpha{:}|[Rc']\langle\alpha : \kappa'\rangle|.\mathcal{R}|c|_{\Theta} : \\ \quad\quad \forall\alpha{:}\kappa'.\widehat{R}\langle\alpha : \kappa\rangle \to |[Rc']\langle\alpha : \kappa\rangle| \to |[Rc']\langle\rho(c) : \kappa\rangle|\end{array}$$

As $\rho(c) = (\lambda\alpha{:}\kappa'.\rho(c))\alpha$, we have produced the correct result type.

□

Lemma 7.6.3 *If* $\Delta;\Gamma \vdash e:\tau$ *then* $\Delta;|\Delta|,|\Gamma| \vdash |e|:|\tau|$

Proof

Proof is by induction on $\Delta;\Gamma \vdash e:\tau$.

case t-tfn

$$[\textit{t-tfn}]\quad \frac{\Delta,\alpha{:}\kappa;\Gamma \vdash e:\sigma}{\Delta;\Gamma \vdash \Lambda\alpha{:}\kappa.e:\forall\alpha{:}\kappa.\sigma}\ (\alpha \notin \mathit{Dom}(\Delta))$$

We need to prove that

$$\Delta;|\Delta|,|\Gamma| \vdash \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\kappa\rangle.|e|:\forall\alpha{:}\kappa.\widehat{R}\langle\alpha:\kappa\rangle \to |\sigma|$$

By induction,

$$\Delta,\alpha{:}\kappa;|\Delta|,x_\alpha{:}\widehat{R}\langle\alpha:\kappa\rangle,|\Gamma| \vdash |e|:|\sigma|$$

case t-tapp

$$\frac{\Delta;\Gamma \vdash e:\forall\alpha{:}\kappa.\sigma \qquad \Delta \vdash c:\kappa}{\Delta;\Gamma \vdash e[c]:\sigma[c/\alpha]}$$

By induction $\Delta;|\Delta|,|\Gamma| \vdash |e|:\forall\alpha{:}\kappa.(\forall\beta{:}\star \to \star.|[R\beta]\langle\alpha:\kappa\rangle|) \to |\sigma|$. Let β be free in c. By Lemma 7.6.1, $\Delta,\beta:\star\to\star;|\Delta| \vdash \mathcal{R}|c|_{(\emptyset,\beta,\bullet)}:|[R\beta]\langle c:\kappa\rangle|$ (note that as β is free in c, we may drop x_β from the term context), so $\Delta;|\Delta| \vdash \Lambda\beta{:}\star\to\star.\mathcal{R}|c|_{(\emptyset,\beta,\bullet)}:\forall\beta{:}\kappa.|[R\beta]\langle c:\kappa\rangle|$, Therefore,

$$\Delta;|\Delta|,|\Gamma| \vdash |e|\ [c]\ \widehat{\mathcal{R}}|c|:|\sigma|[c/\alpha]$$

case t-trec Suppose the term is

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_\oplus:[c']\langle\oplus:\kappa_\oplus\rangle\\ \Delta,\Delta' \vdash c:\kappa\\ \Delta;\Gamma;c' \vdash \Delta' \mid \eta \mid \rho\\ \Delta \vdash c':\star\to\star\end{array}}{\Delta;\Gamma \vdash \mathit{typerec}[\kappa][\Delta',\eta,\rho][c']\ c\ \overline{e}:[c']\langle\rho(c):\kappa\rangle}$$

We want to show that

$$\Delta{:}|\Delta|;|\Gamma| \vdash \mathit{typerec}[\kappa][c']\ \mathcal{R}|c|_{(\emptyset,c',(\Delta',|\eta|,\rho,|\overline{e}|))}\ \overline{e}:|[c']\langle\rho(c):\kappa\rangle|$$

this follows as we may conclude by substitution $\Delta \vdash \rho(c):\kappa$, by the previous lemma, $\Delta;|\Delta| \vdash \mathcal{R}|c|_{(\emptyset,c',(\Delta',|\eta|,\rho,|\overline{e}|))}:|[Rc']\langle\rho(c):\kappa\rangle|$, and by induction $\Delta;|\Delta|,|\Gamma| \vdash e_\oplus:|[c']\langle\oplus:\kappa_\oplus\rangle|$.

□

Table 7.9: Type β-equivalence

Type-β

$$\frac{}{(\Lambda\beta{:}\star \to \star.e)[c] \equiv_{\mathcal{E}} e[c/\beta]}$$

Symmetry

$$\frac{e' \equiv_{\mathcal{E}} e}{e \equiv_{\mathcal{E}} e'}$$

Congruence rules

$$\frac{}{i \equiv_{\mathcal{E}} i} \qquad \frac{}{x \equiv_{\mathcal{E}} x} \qquad \frac{}{R_\oplus \equiv_{\mathcal{E}} R_\oplus}$$

$$\frac{e \equiv_{\mathcal{E}} e'}{\lambda x{:}\sigma.e \equiv_{\mathcal{E}} \lambda x{:}\sigma.e'} \qquad \frac{e_1 \equiv_{\mathcal{E}} e_1' \qquad e_2 \equiv_{\mathcal{E}} e_2'}{e_1 e_2 \equiv_{\mathcal{E}} e_1' e_2'}$$

$$\frac{e \equiv_{\mathcal{E}} e'}{\Lambda\alpha{:}\kappa.e \equiv_{\mathcal{E}} \Lambda\alpha{:}\kappa.e'} \qquad \frac{e \equiv_{\mathcal{E}} e'}{e[c] \equiv_{\mathcal{E}} e'[c]}$$

$$\frac{e \equiv_{\mathcal{E}} e' \qquad e_\oplus \equiv_{\mathcal{E}} e_\oplus'}{\mathit{typerec}[\kappa][c]\ e\ \overline{e} \equiv_{\mathcal{E}} \mathit{typerec}[\kappa][c]\ e'\ \overline{e}'} \qquad \frac{e \equiv_{\mathcal{E}} e' \qquad e_\oplus \equiv_{\mathcal{E}} e_\oplus'}{\mathit{untyrec}[\kappa][c]\ e\ \overline{e} \equiv_{\mathcal{E}} \mathit{untyrec}[\kappa][c]\ e'\ \overline{e}'}$$

7.6.2 Dynamic correctness

We will prove operational correctness up to the definition in Table 7.9 of equivalence of result terms. The symbol $\equiv_{\mathcal{E}}$ relates two LKR terms that differ only by type β-expansions. This notion of equivalence does not weaken our dynamic-correctness result as all equal terms differ only in the type annotations. All equivalent terms have the same erasure, so we can argue that they all model the same computation.

The reason that we can prove operational correctness only up to this notion of equivalence is because of how substitution interacts with the definition of representation. We would like substitution to commute with representation, but that is not the case.

$$\mathcal{R}|c_1[c_2/\alpha]|_{(\Delta,c,\bullet)} \neq \mathcal{R}|c_1|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]$$

For example, if c_1 is α then the left hand side equals $\mathcal{R}|c_2|_{(\Delta,c,\bullet)}$ while the right hand side equals $(x_\alpha[c])[\widehat{\mathcal{R}}|c_2|/x_\alpha] = (\Lambda\beta{:}\star \to \star.\mathcal{R}|c_2|_{(\Delta,\beta,\bullet)})[c]$.

Proposition 7.6.4 *By examination of the definition of $\equiv_{\mathcal{E}}$, we assert the following properties of this relation:*

1. $\equiv_{\mathcal{E}}$ *is an equivalence relation.*

2. *If* $e_1 \equiv_{\mathcal{E}} e_2$ *then* $e[e_1/x] \equiv_{\mathcal{E}} e[e_2/x]$.

3. *If* $e_1 \equiv_{\mathcal{E}} e_2$ *then* $e_1[e/x] \equiv_{\mathcal{E}} e_2[e/x]$.

4. *If* e *is not of the form* $(\Lambda\beta{:}\star \to \star.e_1)[c]$ *and* $e \equiv_{\mathcal{E}} e'$ *then* $e' \mapsto^* e''$ *where* e'' *has the same outermost form as* e *and* $e'' \equiv_{\mathcal{E}} e$.

Lemma 7.6.5 (Weakening) *If* α *is not free in* c*, then for any* Δ, c, c', Ψ*,*

$$\mathcal{R}|c|_{(\Delta,\alpha:\kappa,c',\Psi)} = \mathcal{R}|c|_{(\Delta,c',\Psi)}$$

Proof

Examination of the definition of $\mathcal{R}|c|_{\Theta}$. □

Lemma 7.6.6 (Substitution of closed constructors) *If* $\Delta, \alpha{:}\kappa_2 \vdash c_1 : \kappa_1$ *and* $\emptyset \vdash c_2 : \kappa_2$ *then*

$$\mathcal{R}|c_1[c_2/\alpha]|_{(\Delta,c,\bullet)} \equiv_{\mathcal{E}} \mathcal{R}|c_1|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]$$

Proof

Proof by structural induction on c.

case $c_1 \equiv \oplus$ Trivial.

case $c_1 \equiv \alpha$

$$\begin{aligned}\mathcal{R}|\alpha[c_2/\alpha]|_{(\Delta,c,\bullet)} &= \mathcal{R}|c_2|_{(\Delta,c,\bullet)} \\ &\equiv_{\mathcal{E}} (\Lambda\beta.\mathcal{R}|c_2|_{(\emptyset,\beta,\bullet)})[c] \\ &= \mathcal{R}|\alpha|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\end{aligned}$$

This case relies on the fact that as c_2 is closed then $\mathcal{R}|c_2|_{(\emptyset,c,\bullet)} = \mathcal{R}|c_2|_{(\Delta,c,\bullet)}$.

case $c_1 \equiv \beta$ When $\beta \notin \Delta$,

$$\mathcal{R}|\beta[c_2/\alpha]|_{(\Delta,c,\bullet)} = x_\beta[c] = \mathcal{R}|\beta|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]$$

otherwise, when $\beta \in \Delta$,

$$\mathcal{R}|\beta[c_2/\alpha]|_{(\Delta,c,\bullet)} = y_\beta = \mathcal{R}|\beta|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]$$

case $c_1 \equiv c'c''$

$$\begin{aligned}&\mathcal{R}|c'c''[c_2/\alpha]|_{(\Delta,c,\bullet)}\\&\quad= \mathcal{R}|c'[c_2/\alpha]|_{(\Delta,c,\bullet)}\ [c''[c_2/\alpha]]\ \widehat{\mathcal{R}}|c''[c_2/\alpha]|\ \mathcal{R}|c''[c_2/\alpha]|_{(\Delta,c,\bullet)}\\&\quad\equiv_{\mathcal{E}} (\mathcal{R}|c'|_{(\Delta,c,\bullet)}[c'']\ \widehat{\mathcal{R}}|c''|\ \mathcal{R}|c''|_{(\Delta,c,\bullet)})[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\end{aligned}$$

as by induction

$$\begin{aligned}&\mathcal{R}|c'[c_2/\alpha]|_{(\Delta,c,\bullet)} \equiv_{\mathcal{E}} \mathcal{R}|c'|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\\&(\Lambda\beta.\mathcal{R}|c''[c_2/\alpha]|_{(\emptyset,\beta,\bullet)}) \equiv_{\mathcal{E}} (\Lambda\beta.\mathcal{R}|c''|_{(\emptyset,\beta,\bullet)})[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\\&\mathcal{R}|c''[c_2/\alpha]|_{(\Delta,c,\bullet)} \equiv_{\mathcal{E}} \mathcal{R}|c''|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\end{aligned}$$

case $c_1 \equiv \lambda\beta{:}\kappa'.c'$ This case follows straightforwardly by induction.

$$\begin{aligned}&\mathcal{R}|\lambda\beta{:}\kappa'.c'[c_2/\alpha]|_{(\Delta,c,\bullet)}\\&\quad= \Lambda\beta{:}\kappa'.\lambda x_\beta{:}\widehat{R}\langle\beta:\kappa'\rangle.\lambda y_\beta{:}|[Rc]\langle\beta:\kappa'\rangle|.\mathcal{R}|c'[c_2/\alpha]|_{(\Delta,\beta:\kappa',c,\bullet)}\\&\quad\equiv_{\mathcal{E}} (\Lambda\beta{:}\kappa'.\lambda x_\beta{:}\widehat{R}\langle\beta:\kappa'\rangle.\lambda y_\beta{:}|[Rc]\langle\beta:\kappa'\rangle|.\\&\qquad \mathcal{R}|c'|_{(\Delta,\beta:\kappa',c,\bullet)})[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\\&\quad= \mathcal{R}|\lambda\beta{:}\kappa'.c'|_{(\Delta,c,\bullet)}[c_2/\alpha][\widehat{\mathcal{R}}|c_2|/x_\alpha]\end{aligned}$$

□

Lemma 7.6.7 (Open substitution) *Let* $\Psi = (\Delta', \eta, \rho, \overline{e})$. *If* $\Delta, \alpha{:}\kappa' \vdash c_1 : \kappa$ *and* $\Delta \vdash c_2 : \kappa'$ *then*

$$\mathcal{R}|c_1[c_2/\alpha]|_{(\Delta,c,\Psi)} \equiv_{\mathcal{E}} \mathcal{R}|c_1|_{(\Delta,\alpha:\kappa',c,\Psi)}[\rho(c_2)/\alpha][\widehat{\mathcal{R}}|\rho(c_2)|/x_\alpha][\mathcal{R}|c_2|_{(\Delta,c,\Psi)}/y_\alpha]$$

Proof

Proof by induction on c. For notational convenience, let $\Theta = (\Delta, c, \Psi)$ and let $\Sigma = [\rho(c_2)/\alpha][\widehat{\mathcal{R}}|\rho(c_2)|/x_\alpha][\mathcal{R}|c_2|_\Theta/y_\alpha]$.

case $c_1 \equiv \oplus$ Trivial.

case $c_1 \equiv \alpha$

$$\mathcal{R}|\alpha[c_2/\alpha]|_\Theta = \mathcal{R}|c_2|_\Theta = y_\alpha\Sigma$$

case $c_1 \equiv \beta$ If $\beta \in \Delta$,

$$\mathcal{R}|\beta[c_2/\alpha]|_\Theta = y_\beta = y_\beta\Sigma$$

otherwise if β is bound by a *typerec*, $\beta \in \Delta'$,

$$\begin{aligned}\mathcal{R}|\beta[c_2/\alpha]|_\Theta &= (\mathit{untyrec}[\Delta'(\beta)][c]\ \eta(\beta)\ \overline{e})\\&= \mathcal{R}|\beta|_{(\Delta,\alpha:\kappa,c,\Psi)}\Sigma\end{aligned}$$

otherwise

$$\mathcal{R}|\beta[c_2/\alpha]|_\Theta = x_\beta[c] = x_\beta[c]\Sigma$$

case $c_1 \equiv c'c''$

$$\begin{aligned}&\mathcal{R}|c'c''[c_2/\alpha]|_\Theta\\ &\quad= \mathcal{R}|c'[c_2/\alpha]|_\Theta\ [\rho(c''[c_2/\alpha])]\ \widehat{\mathcal{R}}|\rho(c''[c_2/\alpha])|\ \mathcal{R}|c''[c_2/\alpha]|_\Theta\\ &\quad\equiv_\mathcal{E} (\mathcal{R}|c'|_{(\Delta,\alpha:\kappa,c,\Psi)}\ [\rho(c'')]\ \widehat{\mathcal{R}}|\rho(c'')|\ \mathcal{R}|c''|_{(\Delta,\alpha:\kappa,c,\Psi)})\Sigma\\ &\quad= \mathcal{R}|c'c''|_{(\Delta,\alpha:\kappa,c,\Psi)}\Sigma\end{aligned}$$

As by induction and the previous lemma

$$\widehat{\mathcal{R}}|\rho(c''[c_2/\alpha])| = \widehat{\mathcal{R}}|\rho(c'')|\Sigma$$

case $c_1 \equiv \lambda\beta{:}\kappa.c'$ Follows straightforwardly by induction. The only tricky thing to notice is that $\mathcal{R}|c_2|_{(\Delta,\beta:\kappa',c,\Psi)} = \mathcal{R}|c_2|_{(\Delta,c,\Psi)}$ as β is not free in c_2, by the bound variable convention.

$$\begin{aligned}&\mathcal{R}|\lambda\beta{:}\kappa'.c'[c_2/\alpha]|_{(\Delta,c,\Psi)}\\ &\quad= \Lambda\beta{:}\kappa'.\lambda x_\beta{:}\widehat{R}\langle\beta:\kappa'\rangle.\lambda y_\beta{:}|[Rc]\langle\!|\beta:\kappa'\rangle|.\mathcal{R}|c'[c_2/\alpha]|_{(\Delta,\beta:\kappa',c,\Psi)}\\ &\quad\equiv_\mathcal{E} (\Lambda\beta{:}\kappa'.\lambda x_\beta{:}\widehat{R}\langle\beta:\kappa'\rangle.\lambda y_\beta{:}|[Rc]\langle\!|\beta:\kappa'\rangle|.\\ &\qquad\mathcal{R}|c'|_{(\Delta,\beta:\kappa',c,\Psi)})[\rho(c_2)/\alpha][\widehat{\mathcal{R}}|\rho(c_2)|/x_\alpha][\mathcal{R}|c_2|_{(\Delta,\beta:\kappa',c,\Psi)}/y_\alpha]\\ &\quad= \mathcal{R}|\lambda\beta{:}\kappa'.c'|_{(\Delta,c,\Psi)}\Sigma\end{aligned}$$

□

Lemma 7.6.8 *If* $\Delta \vdash c_1 : \kappa$ *and* $c_1 \leadsto^{wh} c_2$ *then for all* $e_1 \equiv_\mathcal{E} \mathcal{R}|c_1|_{(\Delta,c',\Psi)}$, $e_1 \mapsto^* e_2$ *and* $e_2 \equiv_\mathcal{E} \mathcal{R}|c_2|_{(\Delta,c',\Psi)}$.

Proof

Proof by induction on $c_1 \leadsto^{wh} c_2$.

Suppose $(\lambda\alpha{:}\kappa.c_3)c_4 \leadsto^{wh} c_3[c_4/\alpha]$. Then

$$e_1 \mapsto^* (\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\kappa\rangle.\lambda y_\alpha{:}|[Rc']\langle\!|\alpha:\kappa\rangle|.\mathcal{R}|c_3|_{(\Delta,\alpha:\kappa,c',\Psi)})[\rho(c_4)]\ e_5\ e_6$$

where $e_5 \equiv_\mathcal{E} \widehat{\mathcal{R}}|\rho(c_4)|$ and $e_6 \equiv_\mathcal{E} \mathcal{R}|c_4|_{(\Delta,c',\Psi)}$.

$$\begin{aligned}&(\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\kappa\rangle.\lambda y_\alpha{:}|[Rc']\langle\!|\alpha:\kappa\rangle|.\mathcal{R}|c_3|_{(\Delta,\alpha:\kappa,c',\Psi)})\ [\rho(c_4)]\ e_5\ e_6\\ &\quad\mapsto^* \mathcal{R}|c_3|_{(\Delta,\alpha:\kappa,c',\Psi)})[\rho(c_4)/\alpha]\ [e_5/x_\alpha]\ [e_6/y_\alpha]\\ &\quad\equiv_\mathcal{E} \mathcal{R}|c_3|_{(\Delta,\alpha:\kappa,c',\Psi)})[\rho(c_4)/\alpha]\ [\widehat{\mathcal{R}}|\rho(c_4)|/x_\alpha]\ [\mathcal{R}|c_4|_{(\Delta,c',\Psi)}/y_\alpha]\end{aligned}$$

By the above open substitution lemma, this is η-equivalent to $\mathcal{R}|c_3[c_4/\alpha]|_{(\Delta,c',\Psi)}$.

Otherwise, suppose $c_3 c_4 \rightsquigarrow^{wh} c_3' c_4$. By induction, if $e_3 \equiv_{\mathcal{E}} \mathcal{R}|c_3|_{(\Delta,c',\Psi)}$, then $e_3 \mapsto^* e_3'$ and $e_3' \equiv_{\mathcal{E}} \mathcal{R}|c_3'|_{(\Delta,c',\Psi)}$. So

$$\begin{aligned} e_1 &\mapsto^* e_3\ [\rho(c_4)]\ e_5\ e_6 \\ &\mapsto^* e_3'\ [\rho(c_4)]\ e_5\ e_6 \\ &\equiv_{\mathcal{E}} \mathcal{R}|c_3'|_{(\Delta,c',\Psi)}\ [\rho(c_4)]\ \widehat{\mathcal{R}}|\rho(c_4)|\ \mathcal{R}|c_4|_{(\Delta,c',\Psi)} \\ &= \mathcal{R}|c_3' c_4|_{(\Delta,c',\Psi)} \end{aligned}$$

where $e_5 \equiv_{\mathcal{E}} \widehat{\mathcal{R}}|\rho(c_4)|$ and $e_6 \equiv_{\mathcal{E}} \mathcal{R}|c_4|_{(\Delta,c',\Psi)}$. □

Corollary 7.6.9 *If c weak head normalizes to p, and $e \equiv_{\mathcal{E}} \mathcal{R}|c|_{(\Delta,c,\Psi)}$ then $e \mapsto^* p' \equiv_{\mathcal{E}} \mathcal{R}|p|_{(\Delta,c,\Psi)}$, where p' is a path.*

Lemma 7.6.10 (Operational path correctness) *If*

1. $\emptyset \vdash_\kappa \mathit{typerec}[\kappa][c'][\Delta,\eta,\rho]\ p\ \overline{e} : \sigma$
2. $\mathit{typerec}[\kappa][c'][\Delta,\eta,\rho]\ p\ \overline{e} \Rightarrow_k e$
3. $\overline{e}' \equiv_{\mathcal{E}} |\overline{e}|$
4. $p' \equiv_{\mathcal{E}} \mathcal{R}|p|_{(\emptyset,c',(\Delta,|\eta|,\rho,|\overline{e}|))}$ *is a LKR path*

then

$$\mathit{typerec}[\kappa][c']\ p'\ \overline{e}' \Rightarrow_{HR} e_2 \equiv_{\mathcal{E}} |e|.$$

Proof

Proof by induction on p.

case $p \equiv \alpha$ In this case, p' must be the term $\mathit{untyrec}[\kappa][c']\ e_\alpha\ \overline{e}''$ where $\overline{e}'' \equiv_{\mathcal{E}} |\overline{e}|$ and $e_\alpha \equiv_{\mathcal{E}} |\eta(\alpha)|$. Therefore

$$\mathit{typerec}[\kappa][c']\ (\mathit{untyrec}[\kappa][c']\ e_\alpha\ \overline{e}'')\overline{e}' \Rightarrow_{HR} e_\alpha$$

.

case $p \equiv \oplus$ In this case, p' must be $R_\oplus[c']$ as no other equivalent term is a path. So

$$\mathit{typerec}[\kappa][c']\ R_\oplus[c']\ \overline{e}' \Rightarrow_{HR} e_\oplus' \equiv_{\mathcal{E}} |e_\oplus|$$

case $p \equiv (p_1\ c)$ So p' must be $p'_1\ [c]\ e_c\ e'_c$, where $p'_1 \equiv_{\mathcal{E}} \mathcal{R}|p'|_\Theta$ is a path, $e_c \equiv_{\mathcal{E}} \widehat{\mathcal{R}}|\eta(c)|$ and $e'_c \equiv_{\mathcal{E}} (\mathcal{R}|c|_\Theta)$. Assume $\mathit{typerec}[\kappa][c'][\Delta,\eta,\rho]\ p_1\ \overline{e} \Rightarrow_k e$. By induction $\mathit{typerec}[\kappa][c']\ p'_1\overline{e}' \Rightarrow_{HR} e_1 \equiv_{\mathcal{E}} |e|$. Therefore

$$\begin{aligned}\mathit{typerec}[\kappa][c']\ (p'_1\ [c]\ e_c\ e'_c)\ \overline{e}' &\Rightarrow_{HR} e_1\ [c]\ e_c\ (\mathit{typerec}[\kappa][c']\ e'_c\overline{e}')\\ &\equiv_{\mathcal{E}} |e|\ [c]\ \widehat{\mathcal{R}}|\eta(c)|(\mathit{typerec}[\kappa][c']\ (\mathcal{R}|c|_\Theta)\overline{e})\\ &= |e\ [c]\ (\mathit{typerec}[\kappa'][c'][\Delta,\eta,\rho]\ c\ \overline{e})|\end{aligned}$$

□

Lemma 7.6.11 (Operational typerec correctness) *Let $\Psi = \Delta, \eta, \rho$. If $\mathit{typerec}[\kappa][c'][\Psi]\ c\ \overline{e} \mapsto e$ and $e_1 \equiv_{\mathcal{E}} |\,\mathit{typerec}[\kappa][c'][\Psi]\ c\ \overline{e}|$ then $e_1 \mapsto^* e_2 \equiv_{\mathcal{E}} |e|$.*

Proof

By definition $e_1 \mapsto^* \mathit{typerec}[\kappa][c']\ e_c\ \overline{e}'$, where $e_c \equiv_{\mathcal{E}} \mathcal{R}|c|_{(\emptyset,c',\Psi)}$, and $\overline{e}' \equiv_{\mathcal{E}} |\overline{e}|$.

Proof by induction on κ. If κ is $\star$, then suppose c weak-head normalizes to p. Then by Lemma 7.6.9 $e_c \mapsto^*_{HR} p_c \equiv_{\mathcal{E}} \mathcal{R}|p|_{(\emptyset,c',\Psi)}$. By the previous lemma, path evaluation produces the correct result.

Otherwise, suppose $\kappa \equiv \kappa_1 \to \kappa_2$. Let $e_c \equiv_{\mathcal{E}} \mathcal{R}|c|_{(\emptyset,c',\Psi)}$, and $\overline{e}' \equiv_{\mathcal{E}} |\overline{e}|$ be such that

$$\begin{aligned}e_1 &\mapsto^* \mathit{typerec}[\kappa_1 \to \kappa_2][c']\ e_c\ \overline{e}'\\ &\mapsto \Lambda\beta{:}\kappa_1.\lambda x_\beta{:}\widehat{R}\langle\alpha : \kappa_1\rangle.\lambda y_\beta{:}|[c']\langle\beta : \kappa_1\rangle|.\\ &\qquad \mathit{typerec}[\kappa_1][c'](e_c\ [\beta]\ (\Lambda\gamma.x_\beta[\gamma])\ (\mathit{untyrec}[\kappa_2][c']\ y_\beta\ \overline{e}'))\ \overline{e}'\end{aligned}$$

Let $\Psi' = (\Delta, \alpha{:}\kappa_1, \eta, \alpha{:}y_\beta, \rho, \alpha{:}\beta)$. As

$$\mathit{typerec}[\kappa_1 \to \kappa_2][c'][\Psi]\ c\ \overline{e} \mapsto \Lambda\beta{:}\kappa_1.\lambda y_\beta : [c']\langle\beta : \kappa_1\rangle.\ \mathit{typerec}[\kappa_2][c'][\Psi']\ (c\alpha),$$

we need to show that

$$\begin{aligned}&|\Lambda\beta{:}\kappa_1.\lambda y_\beta{:}[c']\langle\beta : \kappa_1\rangle.\ \mathit{typerec}[\kappa_2][c'][\Psi']\ (c\alpha)\ \overline{e}|\\ &=\ \Lambda\beta{:}\kappa_1.\lambda x_\beta{:}\widehat{R}\langle\beta : k_1\rangle.\lambda y_\beta{:}|[c']\langle\beta : \kappa_1\rangle|.\ \mathit{typerec}[\kappa_2][c']\ \mathcal{R}|(c\alpha)|_{(\emptyset,c',\Psi')}\ \overline{e}\\ &\equiv_{\mathcal{E}} \Lambda\beta{:}\kappa_1.\lambda x_\beta{:}\widehat{R}\langle\beta : \kappa_1\rangle.\lambda y_\beta{:}|[c']\langle\beta : \kappa_1\rangle|.\\ &\qquad \mathit{typerec}[\kappa_2][c']\ (e_c\ [\beta]\ (\Lambda\gamma.x_\beta[\gamma])\ (\mathit{untyrec}[\kappa_1][c']\ y_\beta\ \overline{e}'))\ \overline{e}'\end{aligned}$$

This follows because

$$\begin{aligned}\mathcal{R}|c\alpha|_{(\emptyset,c',\Psi')} &= \mathcal{R}|c|_{(\emptyset,c',\Psi')}\ [\rho(\alpha)]\ \widehat{\mathcal{R}}|\rho(\alpha)|\ \mathcal{R}|\alpha|_{(\emptyset,c',\Psi')}\\ &= \mathcal{R}|c|_{(\emptyset,c',\Psi')}\ [\beta]\ (\Lambda\gamma.x_\beta[\gamma])\ (\mathit{untyrec}[\kappa_2][c']\ (\eta(\alpha))\ |\overline{e}|)\\ &= \mathcal{R}|c|_{(\emptyset,c',\Psi)}\ [\beta]\ (\Lambda\gamma.x_\beta[\gamma])\ (\mathit{untyrec}[\kappa_2][c']\ y_\beta\ |\overline{e}|)\\ &\equiv_{\mathcal{E}} e_c\ [\beta]\ (\Lambda\gamma.x_\alpha[\gamma])\ (\mathit{untyrec}[\kappa_2][c']\ y_\beta\ \overline{e}')\end{aligned}$$

□

Lemma 7.6.12 (Constructor substitution) *If* $\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$ *and* $\Delta \vdash c : \kappa$, *then* $|e[c/\alpha]| \equiv_{\mathcal{E}} |e|[c/\alpha][\widehat{\mathcal{R}}|c|/x_\alpha]$.

Lemma 7.6.13 (Term substitution) *If* $\Delta, ; \Gamma, x : \sigma' \vdash e : \sigma$ *and* $\Delta; \Gamma \vdash e' : \sigma'$, *then* $|e[e'/x]| = |e|[|e'|/x]$.

Lemma 7.6.14 (Operational correctness) *If* $\emptyset \vdash e_1 : \sigma$ *and* $e_1 \mapsto_k e_2$ *then if* $e_1' \equiv_{\mathcal{E}} |e_1|$, $e_1' \mapsto_{HR}^* e_2' \equiv_{\mathcal{E}} |e_2|$.

Proof

Proof by induction on $e_1 \mapsto_k e_2$.

case *ev-β*

$$\frac{}{(\lambda x{:}\sigma.e_3)e_4 \mapsto e_3[e_4/x]}$$

Say $e_1' \equiv_{\mathcal{E}} |(\lambda x{:}\sigma.e_3)e_4|$. So $e_1' \mapsto^* (\lambda x{:}\sigma.e_3')e_4'$ where $|e_3| \equiv_{\mathcal{E}} e_3'$ and $|e_4| \equiv_{\mathcal{E}} e_4'$. This steps to $e_3'[e_4'/x] \equiv_{\mathcal{E}} |e_3|[|e_4|/x]$. By term substitution, this equals $|e_3[e_4/x]|$.

case *ev-app*

$$\frac{e_3 \mapsto e_3'}{e_3 e_4 \mapsto e_3' e_4}$$

Say $e_1' \equiv_{\mathcal{E}} |e_3 e_4| \mapsto^* e_5 e_6$ where $e_5 \equiv_{\mathcal{E}} |e_3|$ and $e_6 \equiv_{\mathcal{E}} |e_4|$. By induction $e_5 \mapsto^* e_5' \equiv_{\mathcal{E}} |e_3'|$. So $e_5 e_6 \mapsto^* e_5' e_6 \equiv_{\mathcal{E}} |e_3'||e_4| = |e_3' e_4|$.

case *ev-ty-β*

$$\frac{}{(\Lambda\alpha{:}\kappa.e)[c] \mapsto e[c/\alpha]}$$

Say $e_1' \equiv_{\mathcal{E}} |(\Lambda\alpha{:}\kappa.e)[c]| \mapsto^* (\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.e')[c]e_c$ where $e' \equiv_{\mathcal{E}} |e|$ and $e_c \equiv_{\mathcal{E}} \widehat{\mathcal{R}}|c|$. This term $\mapsto^* e'[c/\alpha][e_c/x_\alpha] \equiv_{\mathcal{E}} |e|[c/\alpha][\widehat{\mathcal{R}}|c|/x_\alpha]$. By the substitution lemma, this result $\equiv_{\mathcal{E}} |e[c/\alpha]|$.

case *ev-tapp*

$$\frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

Follows from induction.

case e is a *typerec* term. Follows from lemma 7.6.11.

□

$$
\begin{array}{l}
\mathit{fix}\ \mathit{copy} : (\forall\alpha{:}\star.\widehat{R}\langle\alpha:\star\rangle \to T(\alpha\to\alpha)).\\
\quad \Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\rangle.\\
\quad\quad \mathit{typerec}[\star][\lambda\beta.\beta\to\beta]\ (x_\alpha[\lambda\beta.\beta\to\beta])\ \ \mathit{of}\\
\quad\quad\ \ \begin{array}{lll}
\mathit{int} & \Rightarrow & \lambda i{:}\mathit{int}.i\\
\to & \Rightarrow & \Lambda\alpha{:}\star.\lambda r_\alpha{:}T(\alpha\to\alpha).\\
 & & \Lambda\beta{:}\star.\lambda r_\beta{:}T(\beta\to\beta).\\
 & & \quad \lambda f{:}T(\alpha\to\beta).r_\beta\circ f\circ r_\alpha\\
\times & \Rightarrow & \Lambda\alpha{:}\star.\lambda r_\alpha{:}T(\alpha\to\alpha).\\
 & & \Lambda\beta{:}\star.\lambda r_\beta{:}T(\beta\to\beta).\\
 & & \quad \lambda x{:}T(\alpha\times\beta).\langle r_\alpha(\pi_1 x), r_\beta(\pi_2 x)\rangle\\
\mu_\star & \Rightarrow & \Lambda\alpha{:}\star\to\star.\\
 & & \quad \lambda r{:}(\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
 & & \quad\quad \mathit{fix}\ f{:}T(\mu_\star\alpha\to\mu_\star\alpha).\lambda x{:}T(\mu_\star\alpha).\\
 & & \quad\quad\quad \mathit{roll}\ (r\ [\mu_\star\alpha]\\
 & & \quad\quad\quad\quad f\ (\mathit{unroll}\ x))\\
\forall_\star & \Rightarrow & \Lambda\alpha{:}\star\to\star.\\
 & & \quad \lambda r{:}(\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
 & & \quad\quad \lambda x{:}T(\forall_\star\alpha).\\
 & & \quad\quad\quad \Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta:\star\rangle.r[\beta]\ (\mathit{copy}[\beta]\ x_\beta)(x\ [\beta]\ x_\beta)\\
\exists_\star & \Rightarrow & \Lambda\alpha{:}\star\to\star.\\
 & & \quad \lambda r{:}(\forall\beta{:}\star.T(\beta\to\beta)\to T(\alpha\beta\to\alpha\beta)).\\
 & & \quad\quad \lambda x{:}T(\exists_\star\alpha).\\
 & & \quad\quad\quad \mathit{let}\langle\beta,\langle x_\beta,y\rangle\rangle = \mathit{unpack}\ x\ \mathit{in}\\
 & & \quad\quad\quad\quad \mathit{pack}\langle\beta,\langle x_\beta, r[\beta]\ x_\beta\ (\mathit{copy}[\beta]\ x_\beta)\ y\rangle\rangle\\
 & & \quad\quad\quad\quad \mathit{as}\ \exists\beta{:}\star.\widehat{R}\langle\beta:\star\rangle\times\alpha\beta
\end{array}
\end{array}
$$

Figure 7.2: Example: Alternate erasure version of *copy*

7.7 An alternative version

The type-erasure language in this section is complicated by the fact that we must support all code written in LK. However, just as we could define both the primitive recursive *typerec* and the iterative $\mathit{typerec}^{it}$ in Chapter 5, there is an analogous $\mathit{typerec}^{it}$ for this language. Furthermore, all of the examples of Chapter 6 may be written in this simpler language. For example, Figure 7.2 contains *copy* written in this language.

The difference between these two versions is again in the types of the branches of $\mathit{typerec}^{it}$. Like before, if $\oplus$ is of kind $\kappa_\oplus$, the $e_\oplus$ branch of $\mathit{typerec}^{it}$ is of type

$[c']\langle\oplus : \kappa_\oplus\rangle$ instead of $|[c']\langle\oplus : \kappa_\oplus\rangle|$—we do not provide the representations of any type arguments to that branch. For example, in *typerec*^{it}, the $e_\times$ branch is of type $\forall\alpha{:}\star.c'\alpha \to \forall\beta{:}\star.c'\beta. \to c'(\alpha\times\beta)$.

$$[\textit{e-typerec}^{it}]\quad \frac{\begin{array}{l}\Delta \vdash c : \kappa \\ \Delta \vdash c' : \star \to \star \\ \Delta;\Gamma \vdash e_\oplus : [c']\langle\oplus : \kappa_\oplus\rangle \qquad (e_\oplus \in \overline{e}) \\ \Delta;\Gamma \vdash e : [Rc']\langle c : \kappa\rangle\end{array}}{\Delta;\Gamma \vdash \textit{typerec}^{it}[\kappa][c']\ e\ \overline{e} : [c']\langle c : \kappa\rangle}$$

Because we do not need these extra representations during the operation of *typerec*, we do not have to include them in the representations of type constructors. For example, $R_\oplus$ is of type $\forall\beta{:}\star.[R\beta]\langle\oplus : \kappa_\oplus\rangle$ instead of $\forall\beta{:}\star.|[R\beta]\langle\oplus : \kappa_\oplus\rangle|$. Likewise, the syntax for path application does not include the first representation argument.

$$p ::= R_\oplus[c] \mid \textit{untyrec}^{it}[\kappa][c']\ e\ \overline{e} \mid p\ [c]\ e_2$$

When we create the representation of a constructor abstraction or application, we do not pass this argument around.

$$\begin{array}{rcl}\mathcal{R}|\lambda\alpha{:}\kappa.c_1|_\Theta & = & \Lambda\alpha{:}\kappa.\lambda y_\alpha{:}[Rc]\langle\alpha : \kappa\rangle.\mathcal{R}|c_1|_{(\Delta',\alpha:\kappa,c,\Psi)} \\ \mathcal{R}|c_1c_2|_\Theta & = & \mathcal{R}|c_1|_\Theta\ [\rho(c_2)]\ \mathcal{R}|c_2|_\Theta\end{array}$$

Evaluation of path applications or higher-order constructors also omits this representation.

$$[\textit{pv-app0}]\quad \frac{\textit{typerec}^{it}[\kappa_1 \to \kappa][c']\ p\ \overline{e} \Rightarrow_{HR} e'}{\textit{typerec}^{it}[\kappa][c']\ (p\ [c]\ e_2)\ \overline{e} \Rightarrow_{HR0} e'\ [c]\ (\textit{typerec}[\kappa_1][c']\ e_2\ \overline{e})}$$

$$[\textit{ev-trec-arrow0}]\quad \frac{}{\begin{array}{l}\textit{typerec}^{it}[\kappa_1 \to \kappa_2][c']\ e\ \overline{e} \mapsto_{HR} \\ \quad \Lambda\alpha{:}\kappa.\lambda y{:}[c']\langle\kappa : \alpha\rangle. \\ \qquad \textit{typerec}^{it}[\kappa_1][c']\ (e\ [\alpha]\ (\textit{untyrec}^{it}[\kappa_2][c']\ y\ \overline{e}))\ \overline{e}\end{array}}$$

What needs further study is whether anything written with *typerec* may be written with *typerec*^{it} through some sort of pairing operation. If it turns out that *typerec*^{it} is not as expressive, then we must decide whether the limitations in expressiveness are true limitations. I have yet to encounter an example that requires the full capabilities of LKR.

Furthermore, Chapter 5 presented an encoding of the *typerec*^{it} version of LIR, within LU. It also seems interesting to investigate an analogous encoding of LKR with LU.

7.8 Chapter summary

In this chapter, I have developed a type-erasure language supporting higher-order intensional type analysis. The first hurdle was to develop a kind-directed operational semantics for *typerec*, so that we do not need to rely on the syntactic properties of the representations of higher-kinds. This operational semantics draws inspiration from Stone and Harper's language with *singleton kinds* [SH00], which in turn was inspired by Coquand's approach to $\beta\eta$-equivalence for a type theory with Π types and one universe [Coq91]. Because equivalence of constructors in Stone and Harper's language strongly depends on the kind at which they are compared, their procedure drives the kind of the compared terms to the base form before weak-head normalizing and comparing structurally.

The second hurdle with creating the type-erasure language was that we did not want to define a version of reduction for terms with free variables. Instead, we chose to directly replace those variables with a place holder for the result of their interpretation. This place holder draws inspiration from the calculi of Fegaras and Sheard [FS96] and of Trifonov et al. [TSS00]. Fegaras and Sheard designed their calculus to extend iterations to datatypes with function spaces, employing a place holder as the trivial *inverse* of the iterator. Trifonov et al. adapted this idea in a type-level *Typerec* for recursive types. Like the parameterized return constructor of the R-type in this calculus, they parameterize the return kind of a *Typerec* iteration.

Chapter 8

Summary and Directions for Future Research

The ability to represent and analyze compile-time abstractions at run-time is crucial to the implementation of modern systems. This thesis is the first step in a research program to provide a principled basis for such activity and an exploration of the language constructs necessary to support it in a type-safe manner. How do the languages discussed in this thesis contribute to the understanding of run-time type analysis? To answer this question it is important to review the important features of a language that supports such type analysis.

- *It should have a type-erasure semantics.* In order to preserve the distinction between compile-time descriptions and run-time data it is important that the types of the language have no computational effect. In order to support this separation, there must be some sort of dependency between the type language and their term representations. The languages LIR, LXR and LKR each demonstrate how that dependency may be formalized for each of the LI, LX and LH languages.

- *The mechanism for run-time type analysis should be easy to incorporate into the language.* Support for run-time type analysis is of no use if it interferes with other desirable language features. Furthermore, if it is difficult to prove correct and implement, it is not likely to be added to many programming languages. In Chapter 5, I show that it is not the case for *typecase* in the LIR language. In that chapter, I describe how that language may be encoded using higher-order polymorphism. Such polymorphism (at least at the term level) is already a common feature of many programming languages. For example, Cheney and Hinze have used similar techniques to encode type representations into the Haskell language [CH02].

- *All types of the language should be analyzable.* If there is a limitation in what types may be represented, then the reflective programs will suffer in their applicability. For example, the LI language does not allow the analysis of polymorphic, existential, recursive and other types with binding structure. Therefore, even though one can implement dynamic types with LI, terms with such types may not be coerced to that dynamic type. In both Chapter 4 and Chapter 6, I address the problem of analyzing types with binding structure.
- *It should extend to static information beyond the types of the language.* Finally, many programming languages have very rich type constructor languages, and reflecting those constructor languages at run-time is important for many applications. The LX language of Chapter 4 demonstrates how constructor-language datatypes (perhaps representing the type system of a different programming language) may be analyzed at run-time. Likewise, Chapter 6 reflects constructor functions to the term level to be used in the definition of polytypic operations over parameterized data structures.

8.1 Future directions in type analysis

While this thesis represents significant progress in the understanding of run-time type analysis, there are still a few issues that deserve further examination.

8.1.1 Type-level type analysis

The generalization of *typerec* to be an interpreter of the type language in Chapter 6 did not include the type level operator *Typerec*. While it is possible to add LI's *Typerec* (or the more expressive *Typerec* operator of Trifonovet al.[TSS00]), doing so is of limited utility. The purpose of *Typerec* is describe the type of polytypic functions, and if those functions are defined over higher-order constructs, than *Typerec* also needs to be applicable to higher-order constructors. Hinze et al. [HJL02] provide a number of examples of higher-order polytypic term definitions that require higher-order polytypic type definitions. However, adding a *Typerec* operator that may analyze higher-order constructors is difficult. Naively adding a self-interpreter to the typed lambda calculus destroys strong normalization, which means that type equality will be undecidable. However, there are typing mechanisms to prevent non-termination by limiting analysis to functions that themselves do not do analysis. For example, the modal system of Despeyroux et al. [DL01, DPC97] discriminates between parametric and non-parametric functions. Trifonov employ a similar mechanism in LQ [TSS00] in order to analyze recursive types at the type level.

8.1.2 Structural type analysis in practice

While intensional type analysis has traditionally been used in the context of type-based compilation, we would like to incorporate this system in an expressive user language. To do so, we must consider type inference. Furthermore, because this framework depends on a type-passing semantics, it is important to determine its actual run-time cost with respect to compile-time specialization. Finally, because this language supports the analysis of types with binding structure, it may be applicable to adding polytypic programming to object-oriented languages, such as Java. While Java uses the names of classes for dynamic type dispatch, my extension would allow the examination of the structure of the class as well. This would provide a principled basis for reflection, and would allow polytypic operations, such as data-structure traversals, object cloning, and structural equality, to be expressed more concisely.

8.2 Future application areas

In the future, I intend both to continue my study of the foundations of typed programming languages and to apply those results broadly to existing and emerging practical problems. Currently, I am interested in the following application areas.

8.2.1 Type-based program verification

Karl Crary and I have already made contributions in the area of using expressive type systems to specify and verify properties of programs with work on resource bound certification [CW00]. A limiting factor in this line of research is flexibility in the security-policy specification. Currently, the security policy is contained and implied by the specific type system used to type check the program. In order to make this sort of verification feasible we must separate the policy from the type system of the language. Another line of research that must be considered is the trade-off between user annotation of types and automatic type-inference. How much extra information are users willing to add to their code? Yet the more sophisticated we make the type-inference engine (which is in essence an automated theorem prover), the less they will understand the reasons why type inference fails to verify their program.

8.2.2 Extension frameworks for statically-typed languages

The proliferation of domain specific languages has reinforced the idea that there is no perfect language suited for every task. At the same time, programmers are (rightly so) becoming more dependent on sophisticated development environments, debuggers and static checkers to aid their development process. Supporting these new facilities for every new "little language" is quite impossible, so some untyped or dynamically typed languages have included support (in the form of a macro system) for extension. However, the challenges of extending a statically typed language with new type constructs as well as verifying that new term forms always produce well-typed programs have previously prevented the development of similar extension mechanisms.

BIBLIOGRAPHY

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Twenty-Sixth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999.

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Fifth IEEE Symposium on Logic in Computer Science*, pages 95–105, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[AFH94] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *1994 ACM Conference on Lisp and Functional Programming*, pages 12–23, Orlando, June 1994.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, 1997.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Arm97] Joe Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 196–203, 1997.

[Aug98] Lennart Augustsson. Cayenne–a language with dependent types. In *Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, September 1998.

[Bar84] Hendrik P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, second edition, 1984.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Clarendon Press, Oxford, 1992.

[BB85] Corrado Böhm and Allessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BBC+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, INRIA-Rocquencourt, CNRS and ENS Lyon, 1997.

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210, pages 63–81. Springer-Verlag, 1997.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67, 1998.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[BP99a] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[BP99b] Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed *lambda*-calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Los Alamitos, 1991. IEEE Computer Society Press.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, James F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[Car86] Luca Cardelli. Amber. In Guy Coisineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 48–70. Springer-Verlag, 1986.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation, Systems Research Center, November 1989.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[CFJW00] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. *MLton User Guide*, April 2000. `http://www.mlton.org/HTML/main.html`.

[CH02] James Cheney and Ralf Hinze. Poor man's dynamics and generics. Available from `http://www.informatik.uni-bonn.de/~ralf/publications.html`, June 2002.

[CHJ+01] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CK02] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 205–216, Florence, Italy, September 2002. ACM Press.

[Coi87] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 156–167, Orlando, FL, 1987.

[COM02] Microsoft COM technologies, January 2002. `http://www.microsoft.com/com/default.asp`.

[Con82] Robert L. Constable. Intensional analysis of functions and types. Technical Report CSR-118-82, Department of Computer Science, University of Edinburgh, June 1982.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–277. Cambridge University Press, 1991.

[Coq94] Thierry Coquand. A new paradox in type theory. In Dag Prawitz, Brian Skyrms, and Dag Westerståhl, editors, *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science, Uppsala, Sweden, August 7-14, 1991*, volume 134, pages 555–570. Elsevier, Amsterdam, 1994.

[CPM88] Thierry Coquand and Christin Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88 International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66, Tallinn, USSR, December 1988. Springer-Verlag.

[CS98] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pages 201–215. ACM, 1998.

[CW99a] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[CW99b] Karl Crary and Stephanie Weirich. Flexible type analysis (extended version). Technical report, Department of Computer Science, Cornell University, 1999.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, January 2000.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 301–313, Baltimore, MD, September 1998. An extended and revised version of this paper is [CWM02].

[CWM02] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. In *Journal of Functional Programming*, 2002. To appear.

[CZ84] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*, pages 242–257, 1996.

[DL01] Joëlle Despeyroux and Pierre Leleu. Recursion over objects of functional type. *Mathematical Structures in Computer Science*, 11:555–572, 2001.

[DM95] Francois-Nicola Demers and Jacques Malenfant. Reflection and logic, functional and object-oriented programming: a short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.

[DPC97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163, Nancy, France, April 1997. Springer-Verlag.

[dRS84] Jim des Rivieres and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 331–347, Austin, Texas, 1984. ACM Press.

[DRW95] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, January 1995.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semnatics. In Gerard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 280–306. Prentice Hall, 1991.

[Fef62] Solomon Feferman. Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, 27(3):259–316, September 1962.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[Gan86] Robin O. Gandy. An early proof of normalization by A.M. Turing. In J. Roger Seldin and Jonathan R. Hindley, editors, *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts, pages 453–455. Cambridge University Press, Cambridge, 1986.

[GHC02] The GHC Team. *The Glasgow Haskell Compiler User's Guide*, version 5.02 edition, 2002. Available at `http://www.haskell.org/ghc/`.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. dissertation, Université Paris VII, 1972.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GLP00] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. In *2000 ACM SIGPLAN International Conference on Functional Programming*, pages 221–232, Montreal, September 2000.

[Göd31] Kurt Gödel. Über formal unentschiedbare sätze der principia mathematica und verwandter systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–98, 1931.

[Gre98] Dale Green. Trail: The reflection API. In Mary Campione, Kathy Walrath, Alison Huml, and Tutorial Team, editors, *The Java Tutorial Continued: The Rest of the JDK(TM)*. Addison-Wesley Pub Co, 1998. `http://java.sun.com/docs/books/tutorial/reflect/index.html`.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI, February 1995.

[Har01] Robert Harper. Programming languages: Theory and practice. Unpublished, draft available at `http://www.cs.cmu.edu/~rwh/plbook/`, 2001.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[Hin00] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, pages 2–27, Ponte de Lima, Portugal, July 2000.

[HJ00] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the Fourth Haskell Workshop, Montreal, Canada, September 17, 2000*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, August 2000.

[HJL02] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Bernhard Möller Eerke Boiten, editor, *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002)*, pages 148–174, Dagstuhl, Germany, July 2002.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[Hur95] Antonious J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications, TLCA '95*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278, Edinburgh, United Kingdom, April 1995. Springer-Verlag.

[HWC01] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001.

[ISO94] International Organisation for Standardisation and International Electrotechnical Commission. *International Standard ISO/IEC 8652:1995 Ada Reference Manual: Language and Standard Libraries*, 6.0 edition, December 1994.

[ISO98] International Organisation for Standardization and International Electrotechnical Commission. *ISO/IEC 14882:1998 Information Technology–Programming Languages–C++*, September 1998.

[ISO99] International Organisation for Standardization and International Electrotechnical Commission. *ISO/IEC 9899:1999 Programming Languages–C*, December 1999.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. Ph.D. dissertation, Chalmers University of Technology and Gteborg University, 2000.

[Jav02] JavaBeans: The only component for Java technology, May 2002. `http://java.sun.com/products/javabeans/`.

[Jay95] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[JBM98] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[JJ97] Patrick Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Twenty-Fourth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997.

[JJ98] Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

[JJ99] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287, 1999.

[JJ00] Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming (WGP2000), Ponte de Lima, Portugal*. Utrecht University, 2000. Technical Report UU-CS-2000-19.

[Jon92] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.

[Jon97] Mark P. Jones. First-class polymorphism with type inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 15–17 1997.

[JP99] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Available at `citeseer.nj.nec.com/jim97type.html`, 1999.

[JR99] Mark P Jones and Alastair Reid. *Hugs 98 : A functional programming system based on Haskell 98 : User Manual*. Yale Haskell Group and Oregon Graduate Institute of Science and Technology, September 1999. Available at `http://cvs.haskell.org/Hugs/downloads/hugs.pdf`.

[KCJR98] Richard Kelsey, William Clinger, and editors Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, September 1998.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel Bobrow. *The Art of the Meta-Object Protocol*. The MIT Press, 1991.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 419–428, Charleston, South Carolina, 1993.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

[Lam83] Butler Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523, pages 406–426. Springer-Verlag, Berlin, Heidelberg, New York, 1991.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87), (Orlando, FL)*, pages 147–155, October 1987.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, New York, NY, 1997.

[Mee92] Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Men87] Paul Francis Mendler. *Inductive Definition in Type Theory*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, CA, June 1995.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, CA, June 1995.

[MH97] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[Mor95] Greg Morrisett. *Compiling with Types*. Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[MSS01] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 81–91, Snowbird, Utah, June 2001. ACM Press.

[MTC+96] Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Oka99] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 28–35, Paris, France, September 1999.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*, pages 54–67, New York, N.Y., 1996.

[PA93] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *International Conference on Typed Lambda Calculi and Applications*, pages 361–375, 1993.

[PCHS00] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University Computer Science, December 2000.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta, GA, USA, June 1988.

[PH99] Simon Peyton Jones and John Hughes. Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, February 1999. Available from `http://www.haskell.org/definition/`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[PL91] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional programming language. In *FPCA91: Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, NY, August 1991. ACM Press.

[PM93] Christin Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 328–345, 1993. LIP research report 92-49.

[Pou93] Eigil Rosager Poulsen. Representation analysis for efficient implementation of polymorphism. Masters dissertation, DIKU, University of Copenhagen, April 1993.

[PPM90] Frank Pfenning and Christin Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 1990.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265, New York, NY, 1998.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In *Proceedings of the International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer-Verlag, 1984.

[Rue92] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. Ph.D. dissertation, University of Michigan, 1992.

[Rue98] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.*, 1998.

[Ś97] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Twenty-Fourth International Colloquium on Automata, Languages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681, Bologna, Italy, July 1997. Springer-Verlag.

[SDP01] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1–2):1–58, September 2001.

[SH00] Chris Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–225, Boston, MA, USA, January 2000.

[Sha97a] Zhong Shao. Flexible representation analysis. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[Sha97b] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 ACM SIGPLAN Workshop on Types in Compilation*, Amsterdam, June 1997. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[She93] Tim Sheard. Type parametric programming. Technical Report CSE 93-018, Oregon Graduate Institute, 1993.

[Smi84] Brian Cantwell Smith. Reflection and semantics in LISP. In *Fourteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[Sol78] Marvin Solomon. Type definitions with parameters (extended abstract). In *Fifth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–38, Tucson, Arizona, January 1978.

[Ste90] Guy L. Steele Jr. *Common Lisp the Language, 2nd Edition*. Digital Press, 1990.

[Str97] Bjarne Stroustrup. *The C++ Programming Language (Third Edition)*. Addison Wesley Longman, Reading, MA, 1997.

[STS00] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis in type erasure semantics. In *Third ACM SIGPLAN Workshop on Types in Compilation*, Montreal, September 2000.

[SU99] Zdzisław Spławski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 102–113, Paris, France, September 1999.

[Tai67] William W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32:198–212, 1967.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. Ph.D. dissertation, Carnegie Mellon School of Computer Science, 1996. Available as CMU-CS-97-108.

[TMC$^+$96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[Tol94] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *1994 ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, June 1994.

[TSS00] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, September 2000.

[TU96] Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proceedings of the IEEE Symposion on Logic in Computer Science (LICS 96)*, pages 74–85, New Brunswick, New Jersey, 1996. IEEE Computer Society Press.

[Ves97] Møans Vestin. Genetic algorithms in Haskell with polytypic programming. Masters dissertation, Göteborg University, 1997.

[WA99] Daniel C. Wang and Andrew W. Appel. Safe garbage collection = regions + intensional type analysis. Technical report, Princeton, July 1999.

[WA01] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 166–178, Snowbird, Utah, June 2001. ACM Press.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[Wei00] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 58–67, Montreal, September 2000.

[Wei01] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2001.

[Wei02] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *Programming Languages and Systems: 11th European Symposium on Programming, ESOP 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002*, pages 98–114, 2002.

[Wel99] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 306–315, 1988.

[Yan98] Zhe Yang. Encoding types in ML-like languages. In *1998 ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 289–300, Baltimore, MD, September 1998.