The Role of Type Equality in Meta-programming

Emir Pasalic
B.S., The Evergreen State College, 1995

A dissertation submitted to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

September 2004

The dissertation “The Role of Type Equality in Meta-programming” by Emir Pasalic has been examined and approved by the following Examination Committee:

Timothy E. Sheard
Associate Professor
OGI School of Science and Engineering
Thesis Research Adviser

Andrew Black
Professor
OGI School of Science and Engineering

James Hook
Associate Professor
OGI School of Science and Engineering

Andrew Pitts
Professor
University of Cambridge

Walid Taha
Assistant Professor
Rice University

Dedication

To my parents

Acknowledgements

Thanks are first due to my adviser, Tim Sheard without whose help and encouragement none of this would be possible. Acknowledgments and thanks are due to Walid Taha: much of the research presented in Chapter 2 was conducted in collaboration with him and Tim. Also, I wish to thank the members of my thesis committee for their patience and helpful suggestions.

Finally, friends and colleagues whose input over the years has been essential: Zino Benaissa, Bruno Barbier, Iavor Diatchki, Levent Erkök, Bill Harrison and many others.

Contents

Abstract

The Role of Type Equality in Meta-programming

Emir Pasalic

Supervising Professor: Timothy E. Sheard

Meta-programming, writing programs that write other programs, involves two kinds of languages. The meta-language is the language in which meta-programs, which construct or manipulate other programs, are written. The object-language is the language of programs being manipulated.

We study a class of meta-language features that are used to write meta-programs that are statically guaranteed to maintain semantic invariants of object-language programs, such as typing and scoping. We use type equality in the type system of the meta-language to check and enforce these invariants. Our main contribution is the illustration of the utility of type equality in typed functional meta-programming. In particular, we encode and capture judgments about many important language features using type equality. Finally, we show how type equality is incorporated as a feature of the type system of a practical functional meta-programming language.

The core of this thesis is divided into three parts.

First, we design a meta-programming language with dependent types. We use dependent types to ensure that well-typed meta-programs manipulate only well-typed object-language programs. Using this meta-language, we then construct highly efficient and safe interpreters for a strongly typed object language. We also prove the type safety of the meta-language.

Second, we demonstrate how the full power of dependent types is not necessary to encode typing properties of object-languages. We explore a meta-language consisting of the programming language Haskell and a set of techniques for encoding type equality. In this setting we are able to carry out essentially the same meta-programming examples. We also expand the range of object-language features in our examples (e.g.,

pattern matching).

Third, we design a meta-language (called Omega) with built-in equality proofs. This language is a significant improvement for meta-programming over Haskell: Omega's type system automatically manipulates proofs of type equalities in meta-programs. We further demonstrate our encoding and meta-programming techniques by providing representations and interpreters for object-languages with explicit substitutions and modal type systems.

Chapter 1

Introduction

1.1 Meta-programming

Meta-programming is the act of writing programs that generate or manipulate other programs. The programs manipulated are called *object-programs* and are represented as data. The programs doing the manipulation are called *meta-programs.* The language in which meta-programs are written is called a *meta-language.* The language of object-programs is called an *object-language.*

Meta-programming systems can be classified into two broad classes: homogeneous meta-programming systems and heterogeneous meta-programming systems. In homogeneous systems the object and meta-language are the same. In heterogeneous systems, the object- and meta-language are different.

Homogeneous meta-programming languages have received a lot of attention over the years. Several homogeneous meta-programming languages have been implemented [112, 121, 77]. Many issues in homogeneous meta-programming have been thoroughly explored: quasi-quotation [126, 7]; type systems [30, 29, 128, 15, 137]; semantics [10, 82, 128, 129, 86, 108]; intentional analysis [86, 85]; applications [37, 117, 42] and so on.

Programming languages designed specifically to support heterogeneous meta-programming have received less attention. The thesis of this dissertation is that heterogeneous meta-programming can be made into a useful meta-programming paradigm that can provide some of the same benefits as the homogeneous meta-programming paradigm:

1. *Static type safety for heterogeneous meta-programs.* Type safety of heterogeneous meta-programs involves the following. The meta-program is written in a strongly typed language L_M. The object program is written in some object-language L_O, which is also strongly typed, but its type system may be different from the type system of the meta-language. A type-safe heterogeneous meta-program is one that statically guarantees that both the meta-program (in L_M) is type correct, and that any object-language program it generates or analyzes is also type correct (in the type system of L_O).

2. *Semantic invariants.* From the point of view of the meta-programs, object-programs are just data. Often, this means that the values that represent object-programs in a meta-program represent only the (abstract) syntax of object-language programs. In a heterogeneous meta-programming framework the programmer should be given the tools to specify additional invariants that the representation of object-language programs should obey. For example, the meta-program might guarantee that only well-formed, correctly scoped object-programs are constructed.

3. *Practical concerns.* Much of the success of meta-programming languages (e.g., MetaML, Scheme) comes from the abstractions they provide that make common meta-programming tasks easy to write. Such abstractions include quasi-quotation for constructing object-programs, built-in support for renaming of bound variables (hygiene), and so on.

   In a heterogeneous meta-programming language, common tasks such as defining new object-language syntax, parsing, and implementing substitution should be supported by the meta-language. The programming language abstractions that serve as the interface to these common tasks should be intuitive and easy to learn, and should be well integrated with other (non meta-programming specific) features of the meta-language.

1.2 Contributions

We support our thesis by designing a language-based framework for heterogeneous meta-programming. In doing so, we have made a number of specific contributions. Here, we point out the three most significant ones, in order of importance.

First, we illustrate the value of type equality in functional meta-programming languages. We have shown how judgments about many important features of object-languages (such as typing judgments for the simply typed λ-calculus, pattern matching, and box and circle types) can be captured using type equality, and manipulated safely by functional meta-programs. We present detailed descriptions of relevant meta-programming examples as a tutorial intended to demonstrate and teach type-equality based meta-programming.

Second, we show how type equality can be used in an existing functional language (Haskell) and, more importantly, how support for type equality can be built into a sophisticated type system for a practical programming language (called Omega). In Omega, the programmer can use a generalized notion of algebraic data-types conveniently combined with support for type equality to represent interesting judgments about object-language programs. We have implemented a prototype of Omega, and demonstrated its utility on comprehensive heterogeneous meta-programming examples.

Third, we design a programming language with dependent types and support for meta-programming (called MetaD). We use this language to present a novel way of addressing an interesting meta-programming problem (tagless and well-typed interpreters). We also explore the theoretical aspects of such languages by formalizing a core MetaD-like calculus and proving its type safety. We also compare the approach to meta-programming using dependent types in MetaD to the more lightweight approaches using Haskell and Omega.

1.3 Background

We outline some historically relevant work in meta-programming that represents the most direct roots of our own research. In the most general view, meta-programming is ubiquitous in computing. Any program that constructs or manipulates something else that could be considered a program is a meta-program. For example, compilers which translate a program in one object language to programs in another object language are meta-programs. On a more mundane level, even printing to a PostScript printer is meta programming: an application creates a PostScript program based on some internal data-structure and ships this program to the printer, which interprets and executes it to produce a hard copy.

At another level, meta-programming is the study of meta-programs (and meta-languages) as formal systems in their own right. While meta-programming is possible in any programming language that allows for representing data, a number of languages have been designed with abstractions that are intended to make writing meta-programs easier.

The notion of treating programs as data was first explicitly developed by the LISP community. In this context, the notion of *quasi-quotation* [126, 8] was developed as a way of making the interface to the data representing the object program "as much like the object-language concrete syntax as possible [118]." Quasi-quotation is a linguistic device used to construct LISP/Scheme s-expressions that represent LISP/Scheme object programs. The Scheme community has also developed the notion of hygiene [68] to prevent accidental capture and dynamic scoping when manipulating object-language representations that contain binding constructs.

The need for a meta-language (as a programming language that can be used as a common medium for defining and comparing families of (object) languages) was described by Landin [69]. Around the same time, Böhm also proposed using the λ-calculus-based language CuCh as a meta-language for formal language description [12].

Nielson and Nielson [90, 93, 92] define programming languages and calculi that syntactically distinguish meta-level from object-level programs as a part of the language. These two level languages provided a tool

for formulating and studying the semantics of compilation.

Two important meta-programming systems emerged from the study of constructive modal logic by Davies and Pfenning [30, 29]. Davies and Pfenning observed a correspondence between propositions in constructive modal (and temporal) logic and types that can be assigned to certain classes of meta-programs.

The considerable body of research on MetaML [135, 130, 82, 15, 129, 134] described a strongly typed meta-programming language that supports construction and execution of object programs in a strongly typed setting.

Language abstractions that support meta-programming are not limited to functional languages. *Template meta-programming* [36, 37] in C++ (re)uses[1] the notion of a template to perform program generation at compile time. This mechanism has been successfully used in the design and implementation of high-performance C++ libraries [28, 27]. The work cited above just touches the surface of the vast area of meta-programming (Chapter 9 contains a more in-depth discussion of related work), but it illustrates several key ideas that have inspired our research.

Starting with Landin, and throughout the work outlined above, the crucial idea is to approach meta-programming by studying meta-languages as formal systems in their own right. This allows us to concentrate not on any particular meta-program and its properties, but on large classes of meta-programs, and to understand meta-programming at a considerably higher level of abstraction.

The work on quasi-quotation and hygiene in Scheme, MetaML, and even the C++ templates, underscores the importance of thinking clearly about internal object-language representation, and of the interface between the concrete syntax and the internal object-language representation.

The work on logical modalities and type systems (Davies and Pfenning, MetaML) underscores both the utility and the importance of strong and expressive type systems for meta-programming languages.

1.4 A Meta-programming Taxonomy

In this section, we shall outline some basic ways of classifying meta-programs and meta-programming languages. We shall also define some of the vocabulary that will allow us to be precise about distinctions between meta-programming systems. Then, we will examine the “heterogeneous vs. homogeneous” classification in more detail.

[1] It was initially designed as a preprocessing mechanism to add generics to C++.

Generator vs. Analyzer. A basic classification of meta-programs can be expressed in terms of the two broad categories of *program generators* and *program analyzers*[118]. A meta-program is a *program generator* if it only constructs object-language programs based on some inputs. A *program analyzer* is a meta-program that observes (analyzes) an object program, and computes some result. Some meta-programs can be both analyzers and generators, as in the case of source-to-source transformations and optimizations. Some meta-programming languages have meta-programming abstractions for writing of both generators and analyzers (e.g., Scheme, Lisp, $\nu^{\Box}$ [86]), while others (e.g., MetaML or C++ with template meta-programming) only support writing generators.

Homogeneous vs. Heterogeneous. Another way of classifying meta-programs is by dividing them into *homogeneous* and *heterogeneous* [128, 118] meta-programs. This division is based on the identity of the meta- and object- language. A homogeneous meta-program, written in a language L, is a meta-program that constructs or manipulates other L programs. A heterogeneous meta-program is a meta-program, written in a language L_1 that constructs or manipulates object-programs written in some language L_2.

The property of being homogeneous and heterogeneous is closely related to the way that object-programs (considered as data that meta-programs manipulate) are represented in the meta-language. In a weak sense, any programming language with strings can be used to write heterogeneous (or homogeneous) meta-programs, since strings can be used to represent object-programs. However, when we speak about homogeneous meta-programming *languages*, we mean those programming languages that have some built-in data-structures and abstractions that are designed and intended for representing object-programs and are integrated into the larger system.

There is much work in the area of homogeneous meta-programming, in particular, programming languages with special abstractions for writing homogeneous program generators. This work has led to both theoretical breakthroughs and practical benefits.

Open vs. Closed. In our discussion of object-language representation we have touched upon an important design decision faced by the designer of a *meta-language*. Basically, it involves the two following choices:

A closed meta-language. In this situation, the meta-language designer chooses both the meta- and the object- language in advance of any actual meta-programming. The language designer decides on a particular set of linguistic features (e.g., quasi-quotation, typing discipline, hygiene) which are built into the meta-language to allow the programmer to construct object-language programs. A good example of a closed meta-language is MetaML.

The closed-language scenario offers a number of benefits. The meta-language and object-language are

identified once and for all. The *programmer* who uses the meta-language never needs to concern himself with representing, parsing, printing, or even type checking the object-programs. All of these problems can be addressed and solved by the language implementer. This promotes a tremendous amount of reuse across all meta-programs. Moreover, restricting the programmer's access to the underlying representation of object-language programs makes it easier to establish meta-theoretic properties that hold for all object programs. These properties can be used by the compiler/interpreter to perform optimizations, as is the case with the MetaML [121] implementation.

The obvious disadvantage of a closed meta-language manifests itself if the meta-language does not support the object-language the programmer wants to manipulate. For example, MetaML provides the programmer with an excellent way of constructing and executing MetaML object-programs, but if a programmer wants to construct Java programs, he is entirely left to his own devices.

An open meta-language. In this situation, the designer of the meta-language cannot assume what particular object-language the meta-programmer is interested in manipulating. All the language designer can do is to design the meta-language so that it contains useful features that will allow the programmer to encode and manipulate the object language(s) of his choice.

A meta-language can be open with respect to a particular feature of an object language. Many general purpose programming languages do provide some abstractions for encoding object-language syntax. However, most general purpose languages do not provide abstractions for meaningful manipulation of the object-language syntax (e.g., renaming of bound variables, capture-avoiding substitution, and so on). Rather, these operations must be implemented by the programmer for each new object language. This results in a great deal of repeated work across many implementations.

For example, a general-purpose programming language like Standard ML may be open with respect to the programmer's ability to define abstract syntax of new object-languages. Algebraic data-types are a particular mechanism that SML offers to the programmer to accomplish this. Moreover, the type system of SML can guarantee that only syntax-correct object-language terms are ever constructed or manipulated by his meta-programs. SML offers the programmer no comparable abstractions that would allow him to encode sets of well-typed object-language terms. Of course, he can still make sure, by meta-theoretic reasoning about a particular meta-program, that this program manipulates only well-typed expressions. However, the meta-language offers him no guarantee that its type system will reject any meta-programs that try to construct ill-typed object programs.

1.4.1 Homogeneous and Heterogeneous Meta-programming

Homogeneous Meta-programming

A classical example of a homogeneous meta-programming language is Scheme [112]. Here, we present a simple example of such meta-programming. Consider the following two functions written in Scheme. The first function, `sum`, takes a list of numbers and computes their sum. This is a fairly standard functional program involving no meta-programming.

```
(define (sum l)
        (if (null? l) 0 (+ (car l) (sum (cdr l)))))

;; Scheme session transcript
1:=> (sum '(1 2 3))
6
```

The second function, `sumgen`, is quite similar to `sum`, except for the use of Scheme's meta-programming abstractions. Instead of adding the numbers in a list, `sumgen` computes a Scheme program that when executed produces the sum of all the numbers in a list.

```
(define (sumgen l)
        (if (null? l) 0 `(+ ,(car l) ,(sumgen (cdr l)))))

;; Scheme session transcript
1:=> (sumgen '(1 2 3))
(+ 1 (+ 2 (+ 3 0)))
1:=> (eval (sumgen '(1 2 3)))
6
```

Scheme's meta-programming facilities are particularly convenient to work with because programs in Scheme are represented using the same structured expressions as all other data. In Scheme, any expression can be marked by *back-quote*, `(`exp`)`, indicating that the expression should be considered *as constructing an s-expression representing a Scheme program*. Inside a quoted expression, commas `(,exp)` are used as an escape notation. An expression escaped with a comma is evaluated to an s-expression representing a Scheme program, which is then spliced into the larger program, where the comma occurs.

Using these language constructs, the function `sumgen` is a meta-program which acts as a program generator. Given a list of numbers `'(`x_1 x_2 x_3 ... x_n`)`, it constructs a scheme expression `(+` x_1 `(+` x_2 `(+` x_3 ... `(+` x_n `0))))`. Scheme also comes equipped with the construct `eval`, which takes an s-expression representing a Scheme program and executes it. Thus the expression `(eval (sumgen '(1 2 3)))` first generates a program `(+ 1 (+ 2 (+ 3 0)))`, and then evaluates it, returning the result `6`.

Most homogeneous meta-programming languages rely on *quasi-quotation* [8] (e.g., back-quote and comma in Scheme), which can be thought of as a special syntactic interface for constructing object-program code. Some of these languages (e.g., Scheme and MetaML) provide constructs for executing the object-language programs constructed by the meta-program (e.g., `eval` in Scheme and `run` in MetaML).

A drawback of programming in Scheme is that Scheme is not statically typed. First, there is no way of statically guaranteeing type correctness of meta-programs. Second, there is no way of knowing object-programs are well-typed until they are executed by `eval`. For example, consider the following Scheme session:

```
1:=> (define bad-program '(1 2))
bad-program
1:=> bad-program
(1 2)
1:=> (eval bad-program)

*** ERROR:bigloo:eval:
Not a procedure -- 1
#unspecified
```

Using the back-quote notation the programmer is able to construct a nonsensical program `(1 2)`. When we invoke `eval` on it, a runtime error is raised for attempting to apply the number 1 as if it were a function. Static typing in meta-programs has a number of advantages. In addition to guaranteeing that the meta-program encounters no type-errors while manipulating object-programs, a statically typed meta-programming language can also guarantee that any of the object-programs generated by the meta-program are also type-correct. A dissadvantage of these type system is that (in case of meta-programming languages with weaker type systems) they sometime may be too restrictive in object-programs that the programmer is allowed to construct (for an example of this phenomenon see Chapter 2.1.1).

MetaML [128, 137, 129] (and its derivative, MetaOCaml [77]) are examples of statically *typed* homogeneous meta-programming languages. MetaML is designed as a conservative extension of the functional programming language Standard ML [80]. In MetaML, the type system is extended with a special type constructor (called *code*) that is used to classify object programs. For example, a program of type `Int` is a program that produces an integer value. On the other hand, a program of type `(code Int)` is a (meta-)program that produces an object program which, when executed, will produce an integer value.

Let us revisit our `sumgen` example, this time written in MetaML. In MetaML, code brackets (written `< ... >`) play the role of back-quote in Scheme, while tilde (called "escape") is analogous to Scheme's comma operator. The type of code is written with code brackets surrounding a type:

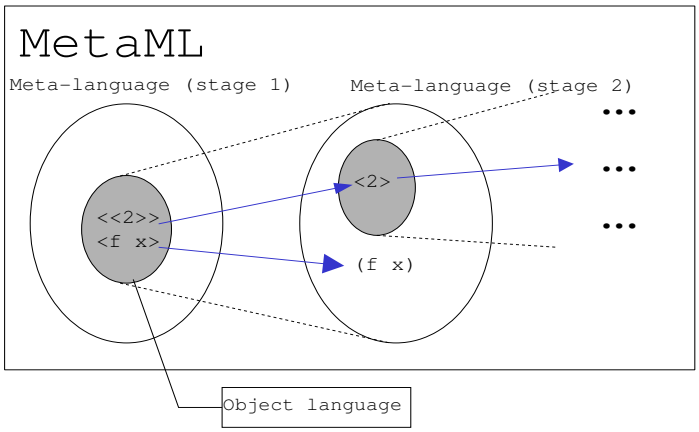


Figure 1.1: Multi-stage structure of a Homogeneous Meta-Language

```
(* sum : int list -> int *)
fun sum [] = 0
  | sum (x::xs) = x + (sum xs)

(* sumgen : int list -> <int> *)
fun sumgen [] = <0>
  | sumgen (x::xs) = < x + ~(sumgen xs) >
```

Many homogeneous meta-programming languages, MetaML included, support *multi-stage programming.* The structure of a multi-stage programming language is illustrated in Figure 1.1. In a multi-stage program, a meta-program can be used to generate an object program which is itself a meta-program generating another program, and so on. The execution time of each meta-program is a *computational stage.* Typed homogeneous meta-programming languages of the MetaML family have three properties that make them well-suited for meta-programming:

1. *Strong typing and type safety [131].* The strong typing of MetaML (also $\lambda^{\Box}$, $\lambda^{\bigcirc}$, and some other statically typed homogeneous meta-languages) guarantees that meta-programs are free from runtime type errors (e.g., adding strings to integers, and so on). Furthermore, their type systems also guarantee that any object programs constructed by a well-typed meta-program will be free of runtime type errors when executed.

2. *Phase errors.* Phase errors occur when an object-language variable is used as if it were a meta-language variable. Consider the following Scheme definition:

```
1:=> (define f1 '(lambda (x) (+ 1 x)))
1:=> f1
(lambda (x) (+ 1 x))
1:=> (define f2 '(lambda (x) ,(+ 1 x)))
*** ERROR:bigloo:+:
```

The first definition, `f1` creates an object program, i.e., a function that adds one to its argument. The second definition, `f2` attempts to create an object program that is a function. However, inside the body of the `lambda` abstraction, the comma operator forces Scheme to evaluate (i.e., in the meta-program) the expression `(+ 1 x)`, where `x` is a variable that is bound only in the object program, and has no value assigned to it. Thus, when trying to evaluate `(+ 1 x)` the Scheme interpreter can find no value for `x`, and raises a runtime exception. If one tries to write `f2` in MetaML, the type checker statically catches such an error:

```
val f2 = <fn x => ~(1+x) >
Error: phase error in 1 + x.
```

3. *Semantic coherence [129].* Object-program code in MetaML is implemented as an abstract data type. This abstract data-type has an important meta-theoretic property, which guarantees that if two MetaML programs, p_1 and p_2, are semantically equivalent, no meta-program can distinguish between their representations as code.

   This property guarantees the soundness of a simple equational theory that can be used to reason about object programs. For example, a program that generates `<(fn x => x) 4>` is equivalent to the program that generates just `<4>`. Since no meta-program can distinguish between those two programs, the MetaML implementation can perform optimizing source-to-source transformations automatically, resulting in the construction of cleaner, more efficient code.

   However, MetaML's semantic coherence has more restrictive consequences. In particular, no meta-program can safely analyze the values of the abstract type that represents object-language programs. The only thing that can safely be done with object-programs, once constructed, is to execute them with `run`. In other words, MetaML only supports the writing of program generators. This prevents the user from implementing a whole class of interesting programs such as syntax-to-syntax transformations (optimizations).

Heterogeneous Meta-programming

In a heterogeneous meta-program, the meta-language and the object-language are different. A typical heterogeneous meta-programming exercise has the following steps:

1. The programmer encodes the syntax of the object language as some form of structured data in the meta-language.

2. The programmer writes a meta-program that

   (a) Constructs an object program, or

   (b) Transforms an existing object- program into another object-program (which may be written in the same object language or not), or

   (c) Computes some other result over an existing object-program, e.g., its meaning, its size, its free variables, its data flow graph, and so on.

How does this scenario compare to meta-programming in the homogeneous setting? When writing a homogeneous meta-program in MetaML the step (1) is unnecessary. The decision about how to represent object-language programs has already been made, once and for all, by the language designer. MetaML provides support for Step (2a) by its strongly typed quasi-quotation. Step (2b) is not directly possible in MetaML, since the language supports only *generative* meta-programming – once constructed, MetaML object programs cannot be analyzed, only executed.

At first glance point (2c) looks like it is not possible in MetaML. But, consider the possibility when the object language is not MetaML, but some other object language represented by an algebraic data-type. Then, an interpreter for this language can be modified so that it computes a residual MetaML program. When run, this program will compute the result more efficiently than simply interpreting the original program [67, 117]. This is a well-known technique often called *staging* an interpreter. We will return to this idea many times later in the dissertation.

1.5 Problem: Object-language Representation

The main problem of designing a useful heterogeneous meta-programming paradigm is the problem of choosing object-language representations. In the next section, we outline a specific set of proposals and approaches to solve this problem. Here, we examine four ways of representing object-programs, and point out the advantages and disadvantages of using each in meta-programming.

Strings. The simplest way of representing object-language programs is to use strings, i.e., to represent object programs *textually*. This technique can be used in both homogeneous and heterogeneous meta-programs, but has an important drawback. Any meta-language with only standard string manipulation

operations (e.g., concatenation, indexing, and so on) offers no abstractions to statically enforce the invariant that strings must represent *only* syntactically correct object-language programs. In practice, object-programs represented as strings may be quite difficult to analyze: some form of parsing must be used to access the underlying structure implicit in the strings. This process is both complicated and error-prone.

One success case with string representations is Perl, a popular programming language for writing CGI scripts. Perl uses a powerful regular expression facility and a number of libraries to make string manipulation of programs more palatable to the programmer.

Algebraic data-types. In functional programming languages, the abstract syntax of object-language programs can be represented using algebraic data-types. Alternatively, other higher-level programming languages have different structured data-facilities such as object hierarchies in Java or s-expressions in Lisp and Prolog. Here, we shall mainly address algebraic data-types in functional languages, but much of the argument should hold for similar data-representations as well.

As a way of representing object-language syntax, algebraic data-types have a major advantage over strings. First, they are a natural way of encoding context-free abstract syntax trees. Consider the following BNF [87] specification of the syntax of a small λ-calculus based language:

$$
\begin{array}{lcl}
\langle \mathsf{Var} \rangle & ::= & x, y, z, \ldots \\
\langle \mathsf{Exp} \rangle & ::= & \langle \mathsf{Var} \rangle \mid \lambda \langle \mathsf{Var} \rangle . \langle \mathsf{Exp} \rangle \mid \langle \mathsf{Exp} \rangle \ \langle \mathsf{Exp} \rangle \\
\langle \mathsf{Decl} \rangle & ::= & \mathsf{let} \ \langle \mathsf{Var} \rangle \ = \langle \mathsf{Exp} \rangle \mid \mathsf{letrec} \ \langle \mathsf{Var} \rangle \ = \ \langle \mathsf{Exp} \rangle
\end{array}
$$

The following Haskell declarations define three data-types, one for each non-terminal specified in the BNF grammar above.

```
type Variable = String
data Exp  = Var Variable
          | Abs Variable Exp
          | App Exp Exp
data Decl = Let Variable Exp
          | LetRec Variable Exp
```

It is not difficult to convince oneself that the three data-types in Haskell represent exactly [2] the parse trees specified by the BNF grammar. It is also important to note that ill-formed syntax trees are statically rejected by Haskell's type system: just as there is no derivation for the ill-formed term $(\mathsf{let}\ x)\ \lambda\ y$, there is no well-typed analogue in Haskell (i.e., the Haskell expression `(App(Let "x") Abs "y")` is rejected by the type-checker).

[2] Modulo undefined values and infinite trees.

The algebraic data-type representation significantly alleviates the draw-backs of string representations. For example, in functional languages pattern matching can be used to analyze algebraic data-types that represent object-programs. The typing discipline of the meta-language like Haskell or Standard ML catches and rejects meta-programs that can build syntactically incorrect object-programs.

However, there are interesting properties of object-programs other than syntactic correctness that are *not* statically enforced by the meta-language using an algebraic data-type representation. For example, context-sensitive properties like typing of object-programs cannot be automatically checked and enforced. For example, a capture avoiding substitution operation for a typed object language should not only produce syntactically well-formed results, but should preserve the type of object-language terms on which it operates. In a meta-program using algebraic data-types, it is up to the programmer to craft his meta-programs so that this meta-theoretic property holds.

Abstract code type. Particularly interesting is the representation for object-language programs used by the homogeneous meta-language MetaML. In a MetaML program, values representing object-language programs are classified by a built-in abstract type of *code*. The programmer constructs and manipulates such values that represent object-language programs using a built-in quasi-quotation mechanism. However, the programmer has no access to the concrete, underlying representation of object-language programs: this representation is chosen by the implementers of MetaML, and fixed once for all.

The MetaML style of code representation has major benefits. First, it statically guarantees that object-language programs represented in this way are syntax-correct and type correct. Second, this representation has several useful meta-theoretic properties: it enforces the correct static scoping discipline; it makes code representations of all $\alpha\beta$-equivalent object programs observationally equivalent to meta-programs. The latter allows the programmer to perform standard equational reasoning about meta-programs in the presence of the `run` construct.

In a heterogeneous setting, a MetaML-style abstract code type is also plausible. However, it is important to note that the choice of how to represent such code internally is a design decision taken by *the language designer* (and implementer), not by the programmer who merely uses the meta-language to write his own meta-programs. Therefore it is less likely to be useful in practice, since one would have to design and implement a new meta-language for every new object-language.

Dependent types. Finally, we describe the most promising approach to representing object-language programs. This particular technique of representation is not new – it has a long history in the logical framework, theorem proving, and type theory community, but has very seldom been used in meta-programming.

This technique is similar to using algebraic data-types to represent the syntax of object-language programs. However, instead of algebraic data-types, it relies on advanced type-theoretic techniques such as the inductive families in the Calculus of Inductive Constructions, predicate encoding in Cayenne [3], or LF [53]. Enriching the meta-language's type system with dependent types allows the programmer to encode not only simple syntactic properties, but semantic ones as well. For example, the programmer can specify a data-type that encodes a set of only syntactically correct *and* type correct syntax trees. All functions that either generate or analyze an object program are forced statically by the meta-language to preserve the semantic properties of the object-language specified by the programmer.

One area of concern when using dependent types is the accessibility and transparency of the type system to the programmer. While a dependent type system can statically enforce object-language program invariants, violating these invariants in meta-programs can result in complex and arcane compilation/type-checking errors that are not easily understood by a novice programmer. Furthermore, to appreciate and use dependent types, one is usually needs considerable background in theoretical computer science and type theory, making dependent types still less accessible to the average programmer.

Generalized algebraic data-types. Using dependent types is an expressive mechanism. We conjecture that properties represented this way can be arbitrarily complex. But, in practice, even very simple properties such as those that enforce correct scoping and typing disciplines of object programs are quite useful. Perhaps something less than the full expressive power of dependent type theory can still be useful in meta-programming?

We will show that this is the case, by devising a method which is an extension of algebraic data-types with the notion of equality between types. We will use this method in two settings. Both of these are sufficient to specify scoping, typing and other invariants of object-language representations.

First, we shall encode type equality in Haskell[3], and use it in conjunction with Haskell's existing algebraic data-types. This technique can be presented to the Haskell programmer as a new programming idiom and is accessible even to Haskell programmers without highly advanced type theoretic background.

Second, we shall design a language (Omega) in which type equality, as a built-in, primitive notion is added to algebraic data-types. Omega provides the programmer with a practical and intuitive interface to type equality leading to smaller programs that are easier to understand and debug than their equivalents in the Haskell setting.

[3]By "Haskel" we actually mean Haskell plus a number of commonly available extensions such as higher-rank polymorphism and existential types, which are available in most popular Haskell implementations.

1.6 Heterogeneous Meta-programming: Desiderata and Approaches

In this section, we shall outline our proposal for putting heterogeneous meta-programming into practice. To do this, we shall have to design a meta-language for heterogeneous meta-programming. We shall outline the requirements, choices, and goals in designing such a meta-language, and the concrete approaches we take to meet them.

There are many useful heterogeneous meta-programs. Recall the example of a postscript printer: the object language is (significantly) different from the meta-language. Another example is a compiler which translates a program in one object language (the input program) into a series of programs in various intermediate languages, finally resulting in a machine-language program.

Left with only general-purpose languages, the programmer must re-implement many heterogeneous meta-programming features from scratch every time he writes a meta-program manipulating a new object language. Moreover, using the abstractions of the meta-language, the programmer has no way to formally check that important semantic invariants of the object-language are preserved. To address these problems we need *an open meta-language for manipulating object-programs that allows for specifying and enforcing their key semantic properties.*

There are a number of goals that such a meta-language should achieve:

1. It must be possible to easily define and manipulate many *different* object languages.
2. It must be possible to express and statically enforce important object language properties such as typing and scoping.
3. It should take into account efficiency, in the sense that the ability to express and manipulate the semantic properties of the object-language should not incur large runtime penalties.
4. It must support good abstraction mechanisms, found in most general purpose-programming languages, for writing software. Such abstraction mechanisms include, but are not limited to recursive definitions, pattern matching, input/output, and so on.
5. It must preserve phase distinction between static type-checking and dynamic (runtime) computation.

Points (1) and (2) are a simple consequence of the fact that we want an *open* meta-language for heterogeneous meta-programming. The meta-language designer has no knowledge of the object-language particulars, but must instead equip the programmer with abstractions and techniques for object-language representation.

They should be *good* techniques and abstractions, or at least better than what's currently offered in general-purpose programming languages: first, the meta-language should be equipped with a type system

that guarantees important semantic properties of object-languages *statically*, where they can be automatically checked and enforced by the meta-language implementation; second, common techniques and programming idioms should be presented to show how the language features of an open meta-language can best be utilized. For example, there should be a clear process of implementing efficient and reliable interpreters for object languages.

The requirement (3) has to do with a general scheme we have for creating implementation of object languages: we shall use staging to make meta-programs such as interpreters highly efficient, applying and extending the technique of staged interpreters [117].

The requirements (4) and (5) have to do with wanting to design a practical programming language – effects (e.g., I/O, imperative features, and so on) must be reconciled with the need to effectively statically type-check meta-programs.

What do we propose to satisfy these requirements? We tried several approaches:

1. We can look for some existing meta-languages that were designed to address other problems and try to use them to solve ours. In fact, several languages used in the logical framework and theorem proving communities (e.g., LF, Coq) seem like good candidates. They allow us to specify type safe heterogeneous meta-programs and to encode semantic properties of object languages. However, in practical terms they leave much to be desired: none of them seem as good candidate for a practical programming language.

2. Lacking an existing meta-language, we can design and implement our own. We shall argue that this is a plausible approach. We describe MetaD, a meta-language we designed to support open heterogeneous meta-programming. We demonstrate the plausibility of MetaD by using it to implement an interesting example of heterogeneous meta-programming. We also present some theoretical results that establish the type safety of a calculus with the same features as MetaD.

   The drawback of this approach is that MetaD is a rather large language with number of advanced programming language features. Adopting MetaD requires many programmers to confront a rather steep learning curve. Implementing, maintaining and promoting such a new language is resource-intensive.

3. We can try combining the approaches (1) and (2). Rather than completely designing a new meta-language from scratch, we can experiment with adding new features to an existing programming language to make it more effective for heterogeneous meta-programming. Of course this approach can be as fraught with complications as the previous approach if we are not careful.

   Fortunately, we read about a new technique for encoding equality proofs in Haskell [143, 4]. This

allowed us to experiment using a set of "tricks" for simulating dependent types in Haskell [75]. We applied these techniques to the problem of representing object-languages with semantic properties and found them highly expressive and very useful. This experimentation was very valuable because it allowed us to test our ideas without getting involved in making any changes to Haskell to start with. We learned much in this process. However, our experience pointed out some practical weaknesses with this approach: constructing Haskell programs that preserve semantic properties of object-language programs is awkward and tedious when using explicit equality proofs since it requires the programmer to explicitly manipulate them at a very low level of abstraction.

4. Experimenting with equality types in Haskell provided motivation for the next step, the design of the Omegalanguage. We were able to add small number of features to Haskell: built-in support for equality types, and inductive kinds. Omega-style equality types allowed us to retain (and even improve upon) the expressiveness of the Haskell-based approach we developed earlier, while making many of the tedious and routine tasks of manipulating equality proofs completely automatic.

In the following sections, we discuss each of these approaches in more detail. We begin by explaining our choice to reject the first approach (Section 1.6.1), and concentrate on the latter two approaches (Sections 1.6.2 and 1.6.3).

1.6.1 Heterogeneous Meta-Programming in Logical Frameworks

Casting about for good candidates for an open heterogeneous meta-language it would not do to overlook a group of formal languages we shall somewhat loosely call logical frameworks [53]. Such meta-languages include various forms of dependently typed calculi [5], and systems such as Twelf [109], and Coq [6]. Implementing programming languages in these systems is based on a powerful idea: use highly expressive types systems with dependent types to represent semantic properties of the object language.

The most important technique is to represent *typing judgments* of the object language as a form of structured data so that only well-typed object programs can be constructed. As we shall see later, this is precisely the technique we shall advocate in the rest of this dissertation. However, from the pragmatic point of view of meta-programming these systems have a number of drawbacks.

1. *They are not designed as real programming languages.* Logical framework-based systems such as Twelf and Coq are mostly targeted at a theorem proving audience. The languages themselves usually have some flavor of dependent typing, and use the Curry-Howard isomorphism to encode logical properties of programs. However, some of these systems (e.g., Alfa [52]) provide only the most rudimentary support for execution of user constructed programs.

Perhaps the most attractive of these languages for open meta-programming is Coq. Although it is a proof checker/theorem prover based on type theory, it is designed to support *extraction* of programs [105]. Extraction allows the user to automatically synthesize a program in Scheme, Haskell or Objective Caml from Coq definitions and theorems. As we demonstrate in Appendix A, this scheme, however, has certain draw-backs of its own: the extracted programs are often type-incorrect (as viewed from the point of view of the extracted-to language). Moreover, the programmer has no direct control over the extraction process and must rely on the implementation of extraction to guarantee the correctness and static safety of generated programs.

More importantly, Coq places considerable restrictions on the programs the user can write: all programs must be guaranteed to terminate,[4] and there is no support for standard programming language features such as I/O or other effects.

Being a consistent proof theory, Coq trades its effectiveness as a programming language to maintain its logical consistency by omitting any programming language features that do not have a pure type-theoretic (logical) meaning. In designing a language for heterogeneous meta-programming, we hope to more evenly balance the requirements of expressiveness with more practical software-engineering concerns.

2. *They are difficult to learn and use by meta-programmers.* In using these systems the programmer must learn a great deal of type theory and logic. This may be inevitable, but perhaps we can find a way to express the necessary type-theoretic and logical concepts in a notation that would be more understandable to a programmer.

3. *They do not address pragmatic concerns such as efficiency.* Efficiency of programming language implementations is an important concern. When semantic properties of object languages are encoded in a meta-language, this encoding may require additional information (such as proofs of these properties) to be constructed and manipulated by the meta-program even when all these properties are static. This often makes meta-programs unnecessarily complex and inefficient.

1.6.2 A Language with Staging and Dependent Types - MetaD

The approach we propose in the first part of the dissertation (Chapter 2) relies on a meta-language with the following features:

1. Dependent types,

[4] This is quite limiting in deriving implementations of object-languages that have recursion or other control features that introduce non-termination.

2. Meta-ML-style staging, and

3. Representation (singleton) types.

We will describe how these features of the meta-language are utilized in heterogeneous meta-programming and how they fit together by presenting a detailed example of meta-programs that manipulate a typed object language. We also prove the type safety of a formalized core (meta-)language that has the features discussed above.

Dependent types. The meta-language we describe in Chapter 2 supports a generalization of algebraic data-types called dependent inductive type families [35].

The semantic properties of object-language syntax, such as object-language typing, are expressed by encoding the typing judgments of the object language as elements of dependent type families. The meta-programs we write manipulate these judgments as well as the syntax of object-language programs. Thus, we assure that whatever manipulations of object programs are performed by the meta-program, only well-typed object programs can be constructed or analyzed.

Meta-ML-style staging. MetaML-style generative meta-programming (also called *staging*) can be very useful in an open heterogeneous meta-language. To illustrate why this should be so, consider implementing an interpreter for some object language L. The programmer first defines a data-type representing the set of expressions of L. Usually, an interpreter maps values from this set of L programs into some set of values V that denote the meanings of L programs.

A standard programming technique relying on MetaML-style staging [117] can be used to improve the interpreter for L in the following ways:

1. Staging can be used to remove the interpretive overhead [67, 117] as a way of generating a more efficient interpreter. First, MetaML meta-programming facilities are be used to divide the interpreter into two stages: the static stage, where the L expressions are analyzed by the interpreter, and the dynamic stage where computation of the interpreted program's value takes place.

   The staged interpreter maps the set of L expressions into a residual meta-language program of type `(code` V`)`. When the staged interpreter is evaluated on some input L-expression, it computes/constructs a residual program is the result of (a) unfolding the interpreter – i.e., removing the case analysis over object programs; (b) removing environment look-ups. Therefore, executing the residual program generated by the staged interpreter for some particular L-expression is significantly more efficient than executing the original, non-staged, interpreter no the same L-expression [48, 67].

2. Moreover, if the object language is strongly typed, the user can define a set of values that represent only well-typed expressions of L. In this case, an interaction between the highly expressive dependent type system and staging can result in an even more efficient staged interpreter by removing the *tagging overhead* [102] that is often present in interpreters written in typed functional programming languages. (We will address this problem in considerable detail in Chapter 2.)

   As we will see, tagging overhead is caused by the type of the use universal value domain V – by replacing the universal value domain V with a dependent type can make tagging unnecessary.

Representation (singleton) types. Finally, we will try to address the issues that arise when combining dependent types and effects such as I/O or non-termination in programming languages. This will require reformulation of the dependently typed meta-language to use *singleton* types [58, 116] – a restricted form of dependent typing.

1.6.3 Haskell as a Heterogeneous Meta-programming Language

The second part of the thesis develops another approach to heterogeneous meta-programming. This approach is primarily motivated by pragmatic considerations. In the first part of the dissertation, we show that introducing a new meta-language with a considerable number of novel features can be used to produce meta-programs that correctly and efficiently manipulate type-correct object-language programs. The second part of the thesis explores the question of whether it is possible that the same (or similar) kind of benefits could be derived in the setting of a functional language like Haskell.

The answer to this question is a qualified "yes." We shall explore how some semantic properties of object-languages can be encoded in the type system of Haskell with commonly available extensions such as existential types and higher-rank polymorphism.

Our approach depends on a technique of encoding equality between types to "fake" dependent and singleton types in Haskell. The only language feature we propose adding to these fairly common extensions of Haskell is *staging* which is essential, we shall argue, for efficient implementations. We shall re-develop the interpreter examples in this new paradigm and compare the two approaches. The comparisons are useful. The techniques are effective, but using them can prove tedious, since they force the programmer to explicitly manage equality proofs in great detail.

1.6.4 Extending Haskell - Omega

We shall adopt an extension to Haskell's type system that makes these techniques significantly easier to use by automating most of the simple, but tedious, equality proof management. We call the resulting meta-language Omega.

We shall also present three further examples of heterogeneous meta-programming. First, we shall define a couple of type-preserving source-to-source transformations on object languages. We shall also extend the range of object-language features presented in Chapters 4 and 5. The goal of this exposition is to provide a kind of meta-programming practicum that can be a source of examples and inspiration to heterogeneous meta-programmers.

1.7 Outline of the Dissertation

The main method of supporting our thesis is demonstration. For meta-programming, we shall concentrate on an interesting class of examples: implementing (staged) interpreters for object-languages. Implementing these interpreters provides the motivation for introduction of the language features and techniques that we design for heterogeneous meta-programming. We demonstrate the open nature of our meta-language by defining and manipulating several different object languages. An important part of the thesis is a tutorial-like presentation that demonstrates how to handle many possible object-language features. We intend this to show how more than just toy object-language features can be incorporated into our framework.

Aside from this introduction chapter, this dissertation is divided into four parts.

- *Part I: Dependent Types for Open Heterogeneous Meta-programming.* In the first part of the dissertation, we define a new meta-language for heterogeneous meta-programming called MetaD. MetaD is a functional language with staging and dependent types. As an example, define a small functional object language, and implement an interpreter for it, demonstrating along the way the benefits derived from the new language features built into MetaD. Next, (Chapter 3) we sketch out a proof of type safety of a simplified core calculus with the same features as MetaD. This proof combines standard syntactic type safety proof techniques [145] with syntactic techniques developed for multi-stage languages [129].

- *Part II: Open Heterogeneous Meta-programming in Haskell.* Rather than implement a meta-language with novel features, we propose a technique for encoding semantic properties of object languages in Haskell.

The key technique that enables heterogeneous meta-programming in Haskell is to replace dependently typed inductive families of MetaD with carefully designed type constructors that encode typing judgments of the object language. We will show how to do this in considerable detail (Chapter 4 describes the general techniques), and implement object-language interpreter similar to the the one used as the main example in Part I (Chapter 5).

- *Part III: Omega and Further Applications.* First, in Chapter 6, we address some of the awkwardness of the Haskell-based techniques introduced in Part II. We do this by proposing a couple of extensions to the type system of Haskell that greatly simplify the writing of typing judgments of the object language. The new language extensions (bundled up in a Haskell-based programming language we call Omega) are presented through several examples. Most interesting of these examples is an implementation of well-typed substitution over simply typed λ-terms, an interesting demonstration of the power of Omega support writing object-language type-preserving syntax-to-syntax transformations.

  Next, we proceed to define and implement meta-programs that manipulate two rather different typed object languages whose type systems are based on modal logic (Chapter 7) and linear-time temporal logic (Chapter 8).

- *Part IV: Conclusion.* First, we survey the relevant related work (Chapter 9). Finally, we summarize our findings, and discuss relevant topic for future work (Chapter 10).

Part I

Dependent Types for Open Heterogeneous Meta-programming

Chapter 2

Meta-programming in a Dependently Typed Framework

2.1 Introduction

In this chapter[1] we begin to explore the design space of heterogeneous meta-programming systems, and to show the advantages of heterogeneous meta-programming over existing approaches to language implementation.

We will begin our exploration by examining the problem of defining tagless interpreters for typed object- and meta- languages. This problem in general is caused by limitations of the type systems of traditional meta-languages.

Type systems of programming languages, especially strongly, statically typed functional languages such as Haskell, are syntactic formal system designed to guarantee the invariant that certain runtime behaviours of programs (runtime type errors such as applying a non-function value) never occur in programs accepted as valid by the type system. In most programming language implementations, the checing of the type validity of programs is performed statically, in a phase prior to execution. In Haskell and ML, to make the type system tractable and amenable to type inference, the type system is designed so that certain programs, even though they do not violate the runtime typing invariants, are nevertheless rejected by the type system.

For example, consider the following function, written in an informal, Haskell-like notation:

```
1 -- f :: Int → Int → ??
2 f 0 x = x
3 f n x = \y → (f (n-1) (x+y))
```

The function `f` takes two integer arguments, `n` and `x` and produces an `n`-ary function that sums those

[1] This chapter is based on material first published in [102].

arguments up. Thus, for example `(f 2 0)` results in the function `\x → \y → x+y+0`.

While the function `f` never causes runtime errors, functional languages such as Haskell or ML reject it because they cannot give it a type: the result type in line 2 is an integer, while the result of the function in the line 3 is a function type that takes an integer argument. In fact, `f` has a whole *family* of function types whose codomain type varies in a regular, predictable way with the value of the function's first argument. Type systems of functional languages such as Haskell do not allow types to depend on values, and reject such functions despite the fact that they can be shown, by meta-theoretical means, never to violate typing discipline at runtime.

In operational terms, what happens when the Haskell type checker tries to infer a type for `f`? First, it tries to infer the result type of the bodies of both branches of the definition of `f`. Then, it attemplts to prove that they are the same type by trying to unify them. However, since it can find no solution to the equation `Int = Int->?`, it rejects `f`.

It is worth noting, however, that the function `f` *can* be given a type in a richer, dependently typed system. Instead returning a result of one particular type, `f` can be seen as returning a result type which depends on the value of the argument `n`:

```
f 0 :: Int → Int → Int
f 1 :: Int → Int → Int → Int
f 2 :: Int → Int → Int → Int → Int
. . .        . . .
f n :: Int → Int → Int → ··· → Int   (n times)
```

Thus, if we could write a function `g` from integers to *types*, we could easily give a type for `f`:

```
g 0 = Int
g n = Int → (g (n-1))

f :: (n:Int) → Int → (g n)
f 0 x = x
f n x = \y → f (n-1) (x+y)
```

Unfortunately, we cannot write such a function `g` in Haskell. If one wanted to implement similar functionality, we would be forced to resort to a more indirect technique.

Recall that the reason why the type-checking in Haskell of the function `f` fails is that for some values of its argument it must return an integer, and for others a function. But Haskell's type system assumes that, no matter what the value of an argument is, the function always returns a result of the same type. A solution to this problem is to use Haskell's data-type facility (combining sum and recursive types in this case) to

produce a type of values that can be either an integer or an n-ary function:

```
data Univ = Z Int
          | S (Int → Univ)
```

Now, we can define a function that encodes the result of `f`, using the data-type `Univ`: `f`:

```
f :: Int → Int → Univ
f 0 x = Z x
f n x = S (\y → f (n-1) (x+y))
```

The type `Univ` is used to unify the two possible kinds of values that `f` computes: integers and functions over integers. The constructors **Z** and **S**, which are there to allow the Haskell type-checker to veryfy that the two cases in the definition of `f` return a value of the same type, also result in runtime behavior of tagging the integer or function values with those constructors.

Now if we apply the function `f` to some integer arguments, e.g., `f 2 0`, it yields a function value equivalent to:

```
S(\a → S(\b → Z(0 + a + b)))
```

We can even define an application operation, which takes the arity of the `Univ` value, the `Univ` value itself, a list of integer arguments to be applied to it (empty if none), and returns the result of the application. The list here serves as another "universal data-type," used to store a (statically unknown) number of arguments to the function encoded by `Univ`. Note that if there is a mismatch between the arity, the number of arguments in the list, and the structure of the `Univ`, a runtime error is raised:

```
applyUniv :: Int → Univ → [Int] → Univ
applyUniv 0 v [] = v
applyUniv n (S f) (arg:args) = applyUniv (n-1) (f arg) args
applyUniv _ _      _         = error "Error in application of Univ"
```

And here is the main difference between the `Univ`-based solution and true dependent types. Whereas the function `f` can be statically type-checked with a dependent type system, the `Univ`-based Haskell solution defers a part of this static type-checking to runtime in form of checking for the tags **S** and **Z**. In other words, whereas we want to statically enforce the invariant that `f` is never applied to the wrong number/type of arguments, Haskell's type system as we have used it here, can only enforce the weaker invariant that `f` is *either* never applied to the wrong number/type of arguments *or* if it is, an error value results at runtime.

In this chapter, we shall concentrate on a very similar problem, that of *superfluous tagging* that often arises in staging and partial evaluation of object language interpreters.

Object Language

```
data Exp =  I Int | Var String | Abs String Exp | App Exp Exp
data Val = VI Int | VF Val->Val

(define mkI   (lambda (i) '(I, i)))
(define mkVar (lambda (x) '(Var, x)))
(define mkAbs (lambda (n e) '(Abs, n, e)))
(define mkApp (lambda (e1 e2) '(App, e1, e2)))
```

Object Language

Metalanguage	Untyped	Typed

Untyped (Scheme)	`(define eval (lambda (t env)`
	` (match-case t`
	`  ((I ?i)  '(VI, i))`
	`  ((Var ?n) (lookUp env n))`
	`  ((Abs ?n ?t0)`
	`   '(VF ,(lambda (v)`
	`          (t0 eval (extend-env env n v)))))`
	`  ((App ?t0 ?t1)`
	`   (match-case (eval t0 env)`
	`    ((VF ?f) (f (eval t1 env)))`
	`    ((?r)  (raise-error "type error"))))`
	`    )))`
		`(define eval (lambda (t env)`
		` (match-case t`
		`  ((I ?i)  i)`
		`  ((Var ?n) (lookUp env n))`
		`  ((Abs ?n ?t0)`
		`    (lambda (v)`
		`     (eval t0`
		`      (extend-env env n v))))`
		`  ((App ?t0 ?t1)`
		`    ((eval t0 env) (eval t1 env)))`
		`     )))`

Typed (Haskell)	`eval e env =`
	` case e of`
	`  I i  →VI i`
	` | Var s  →env s`
	` | Abs (s,e)→`
	`  VF (\v→eval e (ext env s v))`
	` | App f e  →`
	`  case (eval f env) of`
	`  VF vf→vf (eval e env)`
	`  | VI i →error "Runtime type error"`
		`eval e env =`
		` case e of`
		`  I i  →VI i`
		` | Var s  →env s`
		` | Abs (s,e)→`
		`  VF (\v→eval e (ext env s v))`
		` | App f e  →`
		`  case (eval f env) of`
		`  VF vf→vf (eval e env)`
		`  | VI i →error "Impossible case"`

Figure 2.1: Interpreters and Tagging

2.1.1 Superfluous Tagging

Superfluous tagging is a subtle but costly problem that can arise in interpreter implementations when *both* the object- and the meta-language are statically typed. In particular, in most typed meta-languages, there is generally a need to introduce a "universal datatype" (also called "universal domain") to represent object-language values uniformly (see [128] for a detailed discussion). Having such a universal datatype means that we have to perform tagging and untagging operations at the time of evaluation to produce and manipulate object-language values represented by the universal domain.

When the object-language is untyped (or dynamically typed), as it would be when writing a Haskell interpreter for Scheme, the checks *are* really necessary.

When both the the object-language and the meta-language are also statically typed, as it would be when writing an ML interpreter in Haskell, the extra tags are not really needed. They are only necessary to *statically type check the interpreter as a meta-language program.* When this interpreter is staged, it inherits [81] this weakness, and generates programs that contain *superfluous tagging and untagging operations*.

Figure 2.1 provides a brief illustration of this phenomenon. Consider an evaluator for λ-calculus terms, which are defined at the top of the figure: the first two lines represent Haskell data-types encoding the set of expressions and values of the λ-calculus; the bottom four lines are Scheme functions illustrating the structured representations of λ-calculus terms. The bottom half of Figure 2.1 is a table divided into four quadrants, along two dimensions: the horizontal dimension shows whether the object language is statically typed, while the vertical dimension shows whether the meta-language is statically typed. Each quadrant shows a sample implementation of an evaluator: the top row in Haskell (a statically typed meta-language) and the bottom row in Scheme (a dynamically typed meta-language).

In Haskell we use a universal data-type `Val` to represent all the possible values that the evaluator can compute. In the Scheme implementation, we use a particular form of s-expression: integer values are tagged in a list where the head is the atom `'VI` and whose second element is the integer itself; the function values are tagged in a list whose head is the atom `'VF` whose second element is the function value itself.

1. *Untyped Meta-language (Scheme).*

   (a) *Untyped Object Language.* For a dynamically typed object language we must check at runtime whether the value we are applying is indeed a function. If it is not, we must define some semantics of runtime type errors (function `raise-error` in Figure 2.1). We note in passing that it is possible to omit this runtime check, and rely on Scheme's dynamic typing system to catch the error if a value other than a function is applied[2]. However, it is more reasonable to assume that a language designer would want to define her own semantics of runtime type errors.

   (b) *Typed Object Language.* Here, since we can assume (or *know* by meta-theoretical proof) that the object language is statically typed there is no need to implement runtime typing. For a function application we simply evaluate the function expression and the argument expression and then apply the first resulting value to the second.

2. *Typed Meta-language (Haskell)*

   (a) *Untyped Object Language.* Similar to the untyped object language implementation in Scheme, we must introduce tags on runtime values that allow us to check whether what we are applying is indeed a function. We do this with a case analysis on the type `Val`. If the value being applied is not a function, we report a runtime type error.

   (b) *Typed Object Language.* This is surprising: because the object language is strongly typed, we can assume that no type error will occur at runtime (thus, `error "Impossible case"`),

[2] In that case the interpreter would look exactly like 1b.

> and yet still the meta-language (Haskell in this case) forces us to use tags anyway. These tags are a puzzling source of asymmetry – we would expect the Haskell implementation of a statically typed object-language to be a lot more like the one in Scheme.

This asymmetry can be quite costly. Early estimates of the cost of tags suggested that they produce up to a 2.6 times slowdown in the SML/NJ system [133]. More extensive studies in the MetaOCaml system show that slowdown due to tags can be as high as 10 times [16, 62]. How can we remove the tagging overhead inherent in the use of universal value domains?

In the rest of this section we describe the problem of superfluous tags in more detail, and discuss existing approaches to solving it.

An Untyped Interpreter

We begin by reviewing how one writes a simple interpreter in an untyped language. For notational parsimony, we will use Haskell syntax but disregard types. An interpreter for a small lambda language can be defined as follows:

```
data Exp = I Int | Var String | Abs String Exp | App Exp Exp

eval e env =
 case e of
   I i     → i
 | Var s   → env s
 | Abs s e → (\v → eval e (ext env s v))
 | App f e → (eval f env) (eval e env)
```

This provides a simple implementation of object programs represented by the datatype `Exp`. The function `eval` evaluates `e` (an `Exp`) in an environment `env` that binds the free variables in the term to values.

This implementation suffers from a severe performance limitation. If we were able to inspect the result of applying `eval`, such as `(eval (Abs "x" (Var "x")) env0)`, we would find that it is equivalent to

```
(\v→ eval (Var "x") (ext env0 "x" v)).
```

This term will compute the correct result, but it contains an unevaluated recursive call to `eval`. This problem arises in both call-by-value and call-by-name languages, and is one of the main reasons for what is called the “layer of interpretive overhead” that degrades performance [67]. Fortunately, this problem can be eliminated using staging [128].

Staging the Untyped Interpreter

Staging annotations partition the program into (temporally ordered) computational stages so that all computation at stage n is performed before any of the computations at stage $n+1$. Brackets $\langle$ _ $\rangle$ surrounding an expression lift it to the next stage (building code). Escape ~_ drops its expression to a previous stage. The effect of escape is to splice pre-computed code values into code expressions that are constructed by surrounding brackets. Staging annotations change the evaluation order of programs, even evaluating under lambda abstraction. Therefore, they can be used to force the unfolding of the recursive calls to the `eval` function at code-generation time. Thus, by just adding staging annotations to the `eval` function, we can change its behavior to achieve the desired operational semantics:

```
eval' e env =
 case e of
   I i      → ⟨i⟩
 | Var s    → env s
 | Abs s e  → ⟨\v → ~(eval' e (ext env s ⟨v⟩))⟩
 | App f e  → ⟨ ~(eval' f env) ~(eval' e env)⟩
```

Now, applying `eval'` to `(Abs "x" (Var "x"))` in some environment `env0` yields the result $\langle$\v → v$\rangle$. Now there are no leftover recursive calls to `eval'`, since the abstraction case of `eval'` uses escape to evaluate the body of the function "under the lambda:" $\langle$\v → `~(eval' e (ext env s ⟨v⟩))`$\rangle$.

Multi-stage languages come with a run annotation `run _` that allows us to execute such a code fragment. A staged interpreter can therefore be viewed as user-directed way of reflecting a object program into a meta-program, which then can be handed over in a type safe way to the compiler of the meta-language.

Staged Interpreters in a Meta-language with Hindley-Milner Polymorphism

In programming languages, such as Haskell or ML, which use a Hindley-Milner type system, the above `eval` function (staged or unstaged) is not well-typed. Because both integers and functions can be returned as a result of the interpreter, each branch of the case statement may have a different type, and these types cannot be reconciled by simple first order unification.

Within a Hindley-Milner system, we can circumvent this problem by using a "universal type." A universal type is a type that is rich enough to encode values of all the types that appear in the result of a function like `eval`. In the case above, this includes function as well as integer values. A typical definition of a universal type for this example might be:

`data V = VI Int | VF V → V.`

The interpreter can then be rewritten as a well-typed (Haskell) program:

```
unF (VF f) = f
unF (VI _) = error "Tag mismatch, expecting function"

eval e env =
 case e of
   I   i   → I i
 | Var x   → env x
 | Abs x e → F (\v→ eval e (ext env x v))
 | App f e → (unF (eval f env)) (eval e env)
```

Now, when we compute `(eval (Abs "x" (Var "x")) env0)` we get back a value

```
(VF (\v→eval (Var "x") (ext env0 "x" v))).
```

Just as we did for the untyped `eval`, we can stage this version of `eval`:

```
eval e env =
 case e of
    I i     → ⟨VI i⟩
  | Var x   → env x
  | Abs x e → ⟨VF (\v→ ~(eval e (ext env x ⟨v⟩)))⟩
  | App f e → ⟨(unF ~(eval f env)) ~(eval e env)⟩
```

Now computing `(eval (L("x",V "x")) env0)` yields: ⟨`(VF (\v→v))`⟩

Problem: Superfluous Tags

Unfortunately, the result above still contains the tag **VF**. While this may seem like minor issue in a small program like this one, the effect in a larger program will be a profusion of tagging and untagging operations. Such tags would indeed be necessary if the object-language was untyped. But if we know that the object-language is statically typed (for example, as a simply-typed lambda calculus) the tagging and untagging operations are really not needed.

There are a number of approaches for dealing with this problem. Type specialization [63] is a form of partial evaluation that specializes programs based not only on expressions, but also on types. Thus, a universal value domain in an interpreter may be specialized to arbitrary types in the residual versions, removing tags. Another recently proposed possibility is tag elimination [133, 132, 73], a transformation that was designed to remove the superfluous tags in a post-processing phase. Under this scheme, a language implementation is divided into *three* distinct stages (rather than the traditional two, static and dynamic). The extra stage, *tag elimination*, is distinctly different from the traditional partial evaluation (or specialization)

stage. In essence, tag elimination allows us to type check the object program after it has been generated. If it checks, superfluous tags are simply erased from the interpretation. If not, a "semantically equivalent" interface is added around the interpretation. Tag elimination, however, does not *statically* guarantee that all tags will be erased. We must run the tag elimination at runtime (in a multi-stage language). None of the proposed approaches, however, guarantees (at the time of writing the staged interpreter) that the tags will be eliminated before runtime.

We will present an alternative approach that does provide such a guarantee: in fact, the user never introduces the tags in the first place, because the type system of the meta-language is strong enough to avoid any need for them.

2.2 Tagless Interpreters Using Dependent Types

The solution to the tagging problem that we will present is based on the use of a dependently typed multi-stage language as the meta-language in which to implement object languages.

A language has dependent types if its types can depend on values in the program. We have shown an informal example of this in Section 2.1. Crucial to this is the notion of *type families* – collections of related types indexed by a value. A typical dependent type is the dependent product, often written $\Pi x : \tau_1.\tau_2$, where the type τ_2 may depend on the value of the bound variable x. For example, a dependent product $(\Pi x : \mathsf{Int}.\ \mathsf{if}\ x == 0\ \mathsf{then}\ \mathsf{Int}\ \mathsf{else}\ \mathsf{Bool})$ is a type of a function that takes an Integer, and if that Integer is 0, returns another integer; otherwise it returns a Boolean.

We demonstrate this solution by means of an example: we build a compiler from an interpreter [117] by staging, from beginning to end. The slogan we are guided by is *"tag elimination by never introducing the tags in the first place!"*

We start by presenting a definition of a simple, strongly typed, object language, called L_0, giving its syntax and semantics. The remainder of this chapter describes an implementation of a tagless interpreter for L_0 using dependent types and staging.

Dependent types are used to express the invariant that only well-typed object-language programs can be constructed and manipulated by well-typed meta-programs. The interpreter for object-language programs is given family of types that vary with respect to the (object-language) type of the object-language program. For example, this allows it to return function values for object-languages with function types, integer values for object-programs with integer types, and so on. If the object-language type system is designed correctly to exclude object-language programs that "go wrong," then the meta-language type system forces the interpreter to preserve this invariant without needing to check tags to ascertain at runtime whether the execution

of the object-program has indeed "gone wrong."

To illustrate viability of combining dependent types with staging, we have designed and implemented a prototype language we call MetaD. We use this language as a vehicle to investigate the issues that arise when implementing staged language implementations in a dependently typed setting: we thus re-develop the L_0 implementation as a staged interpreter in MetaD. We also discuss the issues that arise in trying to develop a dependently typed programming language (as opposed to a type theory).

For comparison, we give an implementation of a tagless interpreter for L_0 in Coq [139] in Appendix A, where we shall critically examine our Coq implementation and consider its strengths and weaknesses compared to MetaD.

In a subsequent chapter, we will present the technical contribution of formalizing a multi-stage language with such features, and proving its type safety. We do this by capitalizing on the recent work by Shao, Saha, Trifonov and Papaspyrou's on the TL system [116], which in turn builds on a number of recent works on typed intermediate languages [55, 25, 147, 114, 26, 140].

2.2.1 Object-Language Syntax and Semantics

We begin by considering a definition of the syntax and semantics of L_0. L_0 is sufficiently simple to make our development and presentation manageable. It is, however, sufficient to demonstrate the main issues that arise when constructing a tagless interpreter with staging and dependent types. We begin by formally presenting the syntax and semantics of the object language.

$$\begin{array}{lll} \tau & \in \mathbb{T} ::= & N \mid \tau \to \tau \\ \Gamma & \in \mathbb{G} ::= & \langle\rangle \mid \Gamma, \tau \\ e & \in \mathbb{E} ::= & n \mid \lambda\tau.\, e \mid e\ e \mid \mathsf{Var}\ n \end{array}$$

$$\frac{}{\mathsf{EXP}\ \Gamma \vdash n : N}(\text{Nat}) \qquad \frac{\mathsf{EXP}\ \Gamma, \tau \vdash e : \tau'}{\mathsf{EXP}\ \Gamma \vdash \lambda t.e : \tau \to \tau'}(\text{Lam}) \qquad \frac{\mathsf{EXP}\ \Gamma \vdash e_1 : \tau \to \tau' \quad \mathsf{EXP}\ \Gamma \vdash e_2 : \tau}{\mathsf{EXP}\ \Gamma \vdash e_1\ e_2 : \tau'}(\text{App})$$

$$\frac{\mathsf{VAR}\ \Gamma \vdash n : \tau}{\mathsf{EXP}\ \Gamma \vdash \mathsf{Var}\ n : \tau}(\text{Var}) \qquad \frac{}{\mathsf{VAR}\ \Gamma, \tau \vdash 0 : \tau}(\text{Var-Base}) \qquad \frac{\mathsf{VAR}\ \Gamma \vdash n : \tau}{\mathsf{VAR}\ \Gamma, \tau' \vdash (n+1) : \tau}(\text{Var-Weak})$$

Figure 2.2: Syntax and static semantics of of L_0

Syntax. Figure 2.2 defines the syntax and type system of L_0. The language is a version of the simply typed λ-calculus. Types include a base type of natural numbers (N), and function type former ($\to$). For simplicity of the development, we use de Bruijn indices for variables and binders, where natural number indices that identify a variable represent the number of intervening λ-abstractions between the variable's use and binding site.

Type system. The type system of L_0 is presented also in Figure 2.2. It consists of two judgments: the well-typedness judgment defined inductively over the expression, e, ($\mathsf{EXP}\ \Gamma \vdash e : \tau$), and the auxiliary variable judgment, ($\mathsf{VAR}\ \Gamma \vdash n : \tau$), which projects the appropriate type for a variable index from the type assignment Γ. The splitting of the typing rules into two judgments is not essential, but will make our presentation a bit simpler when we define functions by induction on expressions and variable indices, respectively.

$$
\begin{array}{lcl}
\mathcal{T}[\![N]\!] & = & \mathbb{N} \\
\mathcal{T}[\![\tau_1 \to \tau_2]\!] & = & \mathcal{T}[\![\tau_2]\!]^{\mathcal{T}[\![\tau_1]\!]} \\
 & & \\
\mathcal{TA}[\![\langle\rangle]\!] & = & \mathtt{1} \\
\mathcal{TA}[\![\Gamma, \tau]\!] & = & \mathcal{TA}[\![\Gamma]\!] \times \mathcal{T}[\![\tau]\!]
\end{array}
\qquad
\begin{array}{lcl}
[\![\mathsf{EXP}\ \Gamma \vdash e : \tau]\!] & : & [\![\Gamma]\!] \to [\![\tau]\!] \\
[\![\mathsf{EXP}\ \Gamma \vdash n : N]\!]\ \rho & = & n \\
[\![\mathsf{EXP}\ \Gamma \vdash \mathsf{Var}\ n : \tau]\!]\ \rho & = & [\![\mathsf{VAR}\ \Gamma \vdash n : \tau]\!]\ \rho \\
[\![\mathsf{EXP}\ \Gamma \vdash \lambda\tau.e : \tau \to \tau']\!]\ \rho & = & x \mapsto ([\![\mathsf{EXP}\ \Gamma, \tau \vdash e : \tau']\!]\ (\rho, x)) \\
[\![\mathsf{EXP}\ \Gamma \vdash e_1\ e_2 : \tau]\!]\ \rho & = & [\![\mathsf{EXP}\ \Gamma \vdash e_1 : \tau' \to \tau]\!]\rho([\![\mathsf{EXP}\ \Gamma \vdash e_2 : \tau']\!]\rho) \\
 & & \\
[\![\mathsf{VAR}\ \Gamma \vdash n : \tau]\!] & : & [\![\Gamma]\!] \to [\![\tau]\!] \\
[\![\mathsf{VAR}\ \Gamma, \tau \vdash 0 : \tau]\!]\rho & = & \pi_2(\rho) \\
[\![\mathsf{VAR}\ \Gamma, \tau' \vdash (n+1) : \tau]\!]\rho & = & [\![\mathsf{VAR}\ \Gamma \vdash \mathsf{Var}\ n : \tau]\!](\pi_1\rho)
\end{array}
$$

Figure 2.3: Semantics of L_0

Semantics. The semantics of the language L_0 is shown in Figure 2.3. This semantics consists of three parts:

1. The semantics of types, which maps the (syntactic) types of L_0 to their intended meaning, is given as the semantic function $\mathcal{T}[\![\cdot]\!] : \mathbb{T} \to *$ in Figure 2.3. The typing we give the semantic function $\mathcal{T}[\![\cdot]\!]$, $\mathbb{T} \to *$ is purely for reader's convenience. The base sets $\mathbb{N}$, 1, as well as products and function spaces used are set-theoretical entities. For example, the meaning of the type N is the set of natural numbers, while the meaning of the arrow type $\tau_1 \to \tau_2$ is the function space $\mathcal{TA}[\![\tau_2]\!]^{\mathcal{TA}[\![\tau_1]\!]}$. This function's role is to compute that type of the semantic function for expressions (similar to `eval` above), when given that expression's type.

2. The semantics of type assignments are defined as a semantic function $\mathcal{TA}[\![\cdot]\!] : \mathbb{G} \to *$: each type assignment Γ is mapped into a product of the sets denoting the individual types in the assignment. For example, the meaning of the type assignment $\Gamma = \langle\rangle, \mathsf{Int}, \mathsf{Int} \to \mathsf{Int}$, is the product set $(1 \times \mathbb{N}) \times (\mathbb{N} \to \mathbb{N})$. This function's role is to compute the type of the runtime environment of the semantic function, given the particular type assignment under which we type the object-language expressions whose meaning we are trying to compute.

3. Finally, the semantics of programs is defined on *typing judgments*. Given a typing judgment $[\![\mathsf{EXP}\ \Gamma \vdash e : \tau]\!]$, it maps the meaning of the type assignment Γ, $\mathcal{TA}[\![\Gamma]\!]$, to the meaning of the type of the object

expression $\mathcal{T}[\![\tau]\!]$.

The definition of the semantic function $[\![\mathsf{EXP}\ \cdot \vdash \cdot : \cdot]\!] : (\mathsf{EXP}\ \Gamma \vdash e : \tau) \rightarrow \mathcal{TA}[\![\Gamma]\!] \rightarrow \mathcal{T}[\![\tau]\!]$ is given in Figure 2.3 . For its variable case, it uses an auxiliary function which projects (i.e., looks up) that variable's value from the runtime environment: $[\![\mathsf{VAR}\cdot \vdash \cdot : \cdot]\!] : (\mathsf{VAR}\ \Gamma \vdash n : \tau) \rightarrow \mathcal{TA}[\![\Gamma]\!] \rightarrow \mathcal{T}[\![\tau]\!]$.

This is a standard way of defining the semantics of typed languages [138, 51, 110] (also known as *categorical style*), and the implementation in the next section will be a direct codification of this definition.

2.3 A Brief Introduction to Meta-D

In this section we shall enumerate here the main ingredients and features of Meta-D, a meta-language in which we shall then implement the tagless interpreter for L_0. The purpose of this section also is also to familiarize the reader with the syntax and type system of Meta-D, proceeding informally and by example.

Dependent types. In designing Meta-D, we opt for a predicative style of dependent types with universes. The Coq sorts *Set* and *Prop* are unified into a single sort $*_1$, which in turn is classified by an increasing order of sorts $*_2, *_3, \ldots$. All this is fairly standard [2, 139]. This flavor of dependent types, while it works very well in a type-theoretic theorem prover, may introduce some practical problems in a programming language implementation. We will explore how some of these problems may be solved while still maintaining the expressiveness of the type system.

Basic staging operators. The type system of Meta-D includes a modal type constructor $\bigcirc$ (pronounced "code of"), as well as with the standard staging annotations (see Section 2.1.1 for examples of the notation).

Typing rules of the code constructors are fairly standard [30, 29, 128]. The type system prevents phase errors, i.e., prevents uses of values defined at later stages during earlier stages.

Inductive families. Inductive type families (e.g., [33, 22]) can be thought of as dependent data types. While not strictly necessary (one can use Church encodings), they greatly improve the usability of the meta-language.

The syntax for inductive families is largely borrowed from Coq, and has a very similar feel. Syntactically, each data-type defined must first be given its own type (special constants `*1`, `*2` are sorts, where `*1` classifies types, `*2` classifies kinds, and so on). Each constructor's type is written out fully, and is subject to standard *positivity conditions* [6] to ensure that the data-type defined is truly inductive.

For example, to define the inductive family of natural numbers, instead of writing

```
datatype Nat = Z | S of Nat
```

we write

```
inductive Nat : *1 = Z : Nat | S : Nat→Nat.
```

The inductive notation is more convenient when we are defining dependent data-types. Also, it allows the user to define not only inductive types, but also inductive *kinds* (by simply changing, say, $*1$ to $*2$ in the definition). As an example of *dependent* inductive families, consider the following definition:

```
inductive List (a:*1) : Nat→*1 =
   Nil  : List a Z
 | Cons : a→(n:Nat)→(List a n)→(List a (S n))
```

The inductive family `(List a n)` is a family of lists of elements of type `a` with length `n`. After a family's name, `List`, the user lists zero or more *parameters*. A parameter to `List`, in this case, is a type `a : *1`. The parameters are arguments to the type family which do not change in any of the constructors. Next, after the colon, we give the typing of the type family. In the case of `List a`, it is a function from natural numbers, representing the list's length, to the sort of types, $*_1$. Note that `Nat` here is the type of an *index* of the type family. The difference between *parameters* and *indexes* is that while parameters may not be changed anywhere in the types of the constructors, different constructors of the family may vary the values of the *indexes*. For example, the constructor `Cons` takes as its argument the value `a`, and a list of length `n`. The list it constructs, however, has a different index value, namely `S n` indicating that it is one element longer.

To give an example, the list `ex1` below is a list of integers of length three:

```
val ex1 :: List Int (S (S (S Z))) =
       Cons (102 (S (S Z)) (Cons 101 (S Z) (Cons 100 Z Nil)))
```

Values of inductive families can be deconstructed using the `case` construct. The case is designed to be as similar as possible to case expressions in functional programming languages. For example, the following is a map function that converts a list of `a`s to a list of `b`s:

```
fun mapList (a:*1) (b:*1) (f : (a→b)) (n:Nat) (l : (List a n))
           : (List b n)     =
  case l of
     Nil→Nil
   | (Cons x m xs)→(Cons (f x) m (mapList a b f m xs))
```

Dependent products (functions). Functions in Meta-D are defined using an ML-style syntax: `fun funName (arg1:Typ1) ... (argn:Typn) : Typr = ...`. The function name follows the `fun` keyword, and is followed by declarations of the function's arguments, and finally the type of the function's codomain. Function types, $(x{:}t_1)\rightarrow t_2$, (unlike in Coq, they are always written with the arrow

$\rightarrow$) can be dependent, i.e., the codomain type `t2` may mention the variable `x`. Also, the λ-notation is modeled on ML: `fn (x:t)→ e` is an anonymous function that takes an argument of type `t`.

To demonstrate dependent function types, we revisit the example from section 2.1. This involves the function `f`, which takes an argument `n`, some initial integer value `x`, and produces an `n`-ary function that sums up its arguments. The argument `x` is the integer value for the "nullary" case where `n` is zero. First, we define the function `g` which computes the type we can give to `f`:

```
fun g (n:Nat) : *1 =
  case n of
    Z → Nat
  | S n' → (Nat → (g n'))
```

The function `g` takes a natural number n and constructs a type $\overbrace{\texttt{Nat} \rightarrow \cdots \rightarrow \texttt{Nat}}^{n \text{ times}}$.

Now we are define to construct some inhabitants of this type. In particular, the function `f` from Section 2.1 Are there any inhabitants of this type, for a given n? Consider:

```
fun f (n:Nat) (x:Nat) : (g n) =
  case n of
    Z → x
  | S n' → (fn (y:Nat) → (f n' (x+y)))
```

As we have seen, inductive functions can be defined using recursion. It is assumed that the type-checker can prove that recursively defined functions terminate[3].

Another interesting function might be `makeList`, which, given a natural number `n`, produces a list of zeros of the length `n`:

```
fun makeList (n:Nat) : (List Nat n) =
 case n of
  Z → Nil
  S n' → (Cons Z n' (makeList n'))
```

Dependent sums. Dependent sum types are also available. Dependent sum types are written as `[x:t](f x)`. An element of such a sum is a pair of values: the first is a element of type `t`; the second element is of type `f t`, i.e., its type may depend on the value of the first element. The syntax for constructing such a pair

[3] In the actual implementation the user can instruct the type-checker to ignore termination checking, in which case type-checking may not terminate, as in Cayenne. This makes the type system, viewed as logic, unsound, but may be acceptable in programming practice [2]

is written `[x=e1]e2`. Informally, the typing rule for sum introduction may look something like this:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : (\tau_2[x := e_1])}{\Gamma \vdash [x = e_1]e_2 : [x : \tau_1]\tau_2} \text{(Sum)}$$

Dependent sums do not have a special elimination construct. Instead, the user can deconstruct them using `case` expressions and pattern matching.

2.4 A Tagless Interpreter

After that short introduction to the syntax of Meta-D, we can now begin to implement a tagless interpreter for L_0. First, we define data types that represent the syntax of L_0: the basic types, typing environments and expressions. The following non-dependent type families correspond to the syntactic categories of L_0:

```
inductive Typ : *1  = NatT   : Typ
                    | ArrowT : Typ → Typ → Typ
inductive Env : *1  = EmptyE  : Env
                    | ExtE : Env → Typ  → Env
inductive Exp : *1  = EI : Nat → Exp       (* n *)
                    | EV : Nat → Exp       (* Var n *)
                    | EL : Typ → Exp → Exp (* λτ.e *)
                    | EA : Exp → Exp → Exp (* e1 e2 *)
```

Expressions

```
1  inductive J : (Env, Exp, Typ) → *1  =
2         JN : (e1:Env) → (n :Nat) →  J(e1,EI n,NatT)
3       | JV : (e1:Env) → (n:Nat) → (t:Typ) →
4              JV(e1,n,t) → J(e1,EV n,t)
5       | JL : (e1:Env) → (t1:Typ) → (t2:Typ) → (s2:Exp) →
6              J(ExtE e1 t1,s2,t2) → J(e1,EL t1 s2, ArrowT t1 t2)
7       | JA : (e:Env) → (s1:Exp) → (s2:Exp) → (t1:Typ) → (t2 : Typ) →
8              (J(e,s1,ArrowT t1 t2)) → (J(e,s2,t1)) → J(e, EA s1 s2, t2)
```

Variables

```
9  inductive JV : (Env, Nat, Typ) → *1 =
10       | VZ : (e1:Env) → (t:Typ) → JV(ExtE e1 t, Z, t)
11       | VW : (e1:Env) → (t1:Typ) → (t2:Typ) → (i:Nat) → (JV(e1,i,t1)) →
12              (JV(ExtE e1 t2, S i, t1))
```

Figure 2.4: The typing judgment `J` (without representation types)

Next, we implement the type judgment of L_0. To define the judgments, we need a dependent family *indexed* by three parameters: a type assignment `Env`, an expression `Exp`, and a type `Typ`. The relevant definitions are shown in Figure 2.4. Each constructor in this datatype corresponds to one of the rules in the type system for our object language.

We shall examine the various constructors of the inductive family `J` from Figure 2.4 in turn. The basic idea is to use the "judgments as types" principle [53]. We can view a typing rule as a combinator building larger proofs judgments out of smaller ones. Members of the type `J (e,s,t)` are proofs of logical assertions that $e \vdash s : t$. These proofs are built-up using the constructors of the inductive type `J`. These combinators take proofs of hypothesis judgments (and the values of their free variables) to construct the proof conclusion judgment.

1. The rule for natural number constants (`JN`).

   ```
   | JN : (e1:Env)→(n:Nat)→ J(e1,EI n,NatT)
   ```

   Given a type assignment `e1`, and a natural number `n`, we can produce the proof of the typing judgment `J(e1, EI n, NatT)`, i.e., $e_1 \vdash n : N$.

2. The rule for variables. Variables are implemented using the auxiliary judgment `JV`, also an indexed type family, whose indices are the type assignment, a number representing the de Bruijn index of a variable, and the type of the given variable.
   The variable judgment proofs have two cases.

   ```
   inductive JV : (Env, Nat, Typ)→*1 =
     . . .                  . . .              . . .
   ```

   (a) Base case, where the variable index is zero.

   ```
   | VZ : (e1:Env)→(t:Typ)→JV(ExtE e1 t, Z, t)
   ```

   (b) Inductive case (also called *weakening*). Repeated applications of the weakening rule perform the lookup from the environment.

   ```
   | VW : (e1:Env)→(t1:Typ)→(t2:Typ)→(i:Nat)→(JV(e1,i,t1))→
               (JV(ExtE e1 t2, S i, t1))
   ```

3. The rule for lambda abstraction (`Lam`).

   ```
   JL : (e:Env)→(t1:Typ)→(t2:Typ)→(s:Exp)→
        J(ExtE e t1, s, t2)→J(e, EL t1 s, ArrowT t1 t2).
   ```

   In this case, the first argument to the constructor is the type assignment e in which the λ abstraction is typed. Next, it takes two types t_1 and t_2, for the domain and the co-domain of the function expression respectively. Next, it takes an expression s which is the body of the lambda abstraction. Finally, it takes the proof of the antecedent judgment that $(e, t_1 \vdash s : t_2)$, and constructs proof of the judgment $(e \vdash \lambda t.s : t_1 \rightarrow t_2)$. The correspondence between the constructor **`JL`** and the `Lam` rule from Figure 2.2 should be apparent.

```
1  fun project (e:Env) (rho:(envEval e))
2              (n : Nat) (t:Typ)(j : JV(e,n,t)) : (typEval t) =
3    case j of
4      VZ e t→ #2(rho)
5      VW e t1 t2 i j'→(project e (#1(rho)) i t1 j')
6
7  fun eval (e : Env) (rho: envEval(e))
8           (s : Exp) (t : Typ) (j : J(e,s,t)) : (typEval t) =
9    case j of
10     JN  e n →n
11   | JV  e n t jv →project e rho n t jv
12   | JL e t1 t2 s2 j'→
13       (fn v:(typEval t1)→(eval (ExtE e t1) (rho,v) s2 t2 j'))
14   | JA e s1 s2 t1 t2 j1j2→
15         (eval e rho s1 (ArrowT t1 t2) j1) (eval e rho s2 t1 j2)
```

Figure 2.5: Dependently typed tagless interpreter

4. The rule for application (JA) implements the App rule from Figure 2.2: given two premises $e \vdash s_1 : t_1 \rightarrow t_2$ and $e \vdash s_2 : t_1$, the constructor builds the conclusion $e \vdash s_1\ s_2 : t_2$:

```
| JA : (e:Env)→(s1:Exp)→(s2:Exp)→(t1:Typ)→(t2 : Typ)→
       J(e,s1,ArrowT t1 t2)→J(e,s2,t1)→
       J(e, EA s1 s2, t2)
```

In the definition of Js we see differences between the traditional datatype definitions and inductive datatypes: each of the constructors can have dependently typed arguments and a co-domain type J whose index arguments are different. Data-types in functional languages, on the other hand, force the constructors to return always the *same* type of the result. The variability of inductive family indexes will allow us to define functions by cases in which each case produces or consumes a value of the same inductive type, but where each case differs in the values of the indexes.

The definition of J allows us to use this variability in the indices to enforce the following invariant: given a type assignment e, and an object-language expression s, and an object-language type t, if we can construct an inhabitant (proof) of the judgment J(e,s,t), then $e \vdash s : t$ (in the sense of the Figure 2.2). No functions that manipulate and produce proofs of typing judgments J can break this invariant and remain well-typed in MetaD.

2.4.1 Interpreters of Types and Judgments

Having defined L_0 typing judgments as Meta-D inductive families, we are ready to implement the L_0 interpreter in form of the function eval from Figure 2.5. One thing to note, however, is that the type of the range of the function eval must depend on the L_0 type of the judgment being interpreted: for an integer L_0 program, the result will be an integer, for a function L_0 program, it will be a function and so on.

This dependency is captured in the interpretation function `typEval`. Recall that syntax of L_0 are represented by inductive family `Typ`. The function `typEval` gives the meanings of these object language types by mapping the inductive family `Typ` into meta-language types $*_1$:

```
1 fun typEval (t : Typ) : *1 =
2     case t of
3       NatT → Nat
4     | ArrowT t1 t1 → (typEval t1) → (typEval t2)
```

Predictably, `typEval` maps L_0 type `NatT` to the meta-language type of natural numbers `Nat`. Similarly, given a L_0 arrow type `ArrowT t1 t2`, `typEval` computes a Meta-D arrow type `(typEval t1)` → `(typEval t2)` as its meaning (line 4, above).

Similarly, *type assignments* of L_0 must be given a meaning as well, since the type judgments of L_0 programs depend on the structure of the type assignments which give types for the free variables in the L_0 expressions. Recall that L_0 type assignments are represented by the inductive family `Env`: its structure is that of a list of L_0 types.

The function `envEval` takes a representation of the L_0 type assignment and computes the Meta-D type of the runtime environments corresponding to that type assignment:

```
1 fun envEval (e : Env) : *1 =
2     case e of
3        EmptyE  → unit
4     |  ExtE e2 t → (envEval e2, typEval t)
```

The runtime environment corresponding to the empty type assignment is simply the unit type. For a type assignment `e2` extended by the type `t`, `ExtE e2 t`, the type of the runtime environment is the product of the meaning of `e2` and the meaning of `t`: `(envEval e2, typEval t)`.

The function `eval` is defined by case analysis on the proofs of the typing judgments (Figure 2.5). There are four such cases, each of which we shall examine in some detail:

```
fun eval (e:Env) (rho: envEval(e)) (s:Exp) (t:Typ) (j:J(e,s,t)) : (typEval t) =
```

1. **Natural number literals.** The first case is the judgment for L_0 literal expressions. If the proof of the judgment `j` of type `J(e,s,t)` is of the form **`JN e n`**, then by the definition of `J`, we know that the expression `s` is of the form **`EI n`**, and that the L_0 type `t` is equal to **`NatT`**. The codomain type of `eval` is `typEval t`, but since `t` equals **`NatT`**, we know that the result type of this case branch must be `typEval NatT`, which is equal to the Meta-D type `nat`. Fortunately, we have a `nat`, namely, `n`.

   ```
   fun eval (e:Env) (rho: envEval(e)) (s:Exp) (t:Typ) (j:J(e,s,t)) : (typEval t) =
     case j of
   ```

```
  JN e n→n
  . . .    . . .    . . .
```

2. **Variables.** The variable case is more interesting. First, note that in this branch, the expression index is `EV n` for the natural number `n` which represents the index of the variable expression. The constructor **JV** carries a proof of the variable sub-judgment `JV(e,n,t)`.

```
fun eval (e:Env) (rho: envEval(e)) (s:Exp) (t:Typ) (j:J(e,s,t)) : (typEval t) =
  case j of
    JV e n t jv→project e rho n t jv
    . . .    . . .    . . .
```

Thus, the meaning of variable judgments relies on the auxiliary function `project`, which implements the meaning of variable judgments:

```
fun project (e:Env) (rho:(envEval e)) (n:Nat) (t:Typ) (j:JV(e,n,t)) : (typEval t) =
  case j of
    VZ e t→ #2(rho)
    VW e t1 t2 i j'→(project e (#1(rho)) i t1 j')
```

The function `project` is defined by cases on the inhabitants of the variable judgment `JV(e,n,t)`, where `n` is the natural number index of the variable expression. There are two cases

(a) The base case where the natural number index is zero. In this case, we know that `e` is of the form `(ExtT e' t)`. We also know that the type of the runtime environment `rho` is `envEval (ExtT e' t)` which is equivalent to the pair `(evalEnv e',evalTyp t)`. Now, to produce the result of `typEval t`, all we have to do is project the second element of the pair `rho`.

```
fun project (e:Env) (rho:(envEval e)) (n:Nat) (t:Typ)
            (j:JV(e,n,t)) : (typEval t) =
  case j of
    VZ e' t→ #2(rho)
     . . .    . . .
```

(b) The case where the index is greater than zero. Thus, the index `n` is equal to `S m`. We also know that the type assignment `e` is of the form `(ExtE e' t2)`, and that we have the sub-judgment `j'` of type `JV(e',m,t)`. Furthermore, runtime environment `rho` is of the type `(envEval (ExtE e' t2))` which is just a pair `(envEval e', envEval t2)`.

Recall that the result we are computing is of the type `(typEval t)`. This result can be obtained by projecting `m`-th variable from the sub-judgment `j'` under the first element of `rho`:

```
fun project (e:Env) (rho:(envEval e)) (n:Nat) (t:Typ) (j:JV(e,n,t))
                        : (typEval t) =
  case j of
     . . .    . . .    . . .
   | VW e' t t2 m j'→(project e' (#1(rho)) m t j')
```

3. **Abstractions.**

```
fun eval (e:Env) (rho:envEval(e)) (s:Exp) (t:Typ) (j:J(e,s,t)) : (typEval t) =
  case j of
     . . .
  | JL e t1 t2 s2 j' →
      (fn v:(typEval t1)→(eval (ExtE e t1) (rho,v) s2 t2 j'))
```

   In the case for abstraction judgments we know the following:

   (a) `s` = **`EL`** t_1 s_2

   (b) `j'` : `J(`**`ExtE`** `e` t_1, s_2, t_2`)`

   (c) `t` = **`ArrT`** t_1 t_2

   (d) The result type `typEval t` is `typEval (`**`ArrT`** t_1 t_2`)`, which is equal to `(typEval` t_1`)`$\rightarrow$`(typEval`

   Thus, the value that we are constructing in this branch must be of a function type `(typEval` t_1`)->(typEval` t_2`)`: we λ-abstract over a variable `v : (typEval` t_1`)`, and must produce a value of type `(typEval` t_2`)`. Fortunately, we can do this if we evaluate recursively the proofs of the sub-judgment `j'`. This `j'` must be evaluated in an extended runtime environment of type `(envEval e,typEval` t_1`)`, which we can construct by pairing `rho` with `v`.

4. **Applications.** Evaluating proofs application judgments is straightforward.

```
fun eval (e:Env) (rho:envEval(e)) (s:Exp) (t:Typ) (j:J(e,s,t)) : (typEval t) =
  case j of
     . . .
  | JA e s1 s2 t1 t2 j1j2 →
        (eval e rho s1 (ArrowT t1 t2) j1)  (eval e rho s2 t1 j2)
```

   The judgment proof `j` is constructed from two sub-proofs:

   (a) j_1 : `J(e,`s_1`,`**`ArrT`** t_1 t_2`)`

   (b) j_2 : `J(e,`s_2`,`t_1`)`

   Recall that the value we are trying to compute is of the type t_2. Recursively evaluating j_1 gives us a function of type `(typEval` t_1 $\rightarrow$ `typEval` t_2`)`. Recursively evaluating j_2 gives us a value of type type `typEval` t_1. Simply applying the former to the latter yields the required result.

To review, the most important feature to note about the function `eval` is that writing it does not require that we use tags on the result values, because the type system allows us to specify that the return type of this function is `typEval t`. Tags are no longer needed to help us discriminate what type of value we are getting back at runtime: the type system now tells us, *statically*.

2.4.2 Staged Interpreters in Meta-D

Figure 2.6 shows a staged version of `eval`. As with Hindley-Milner types, staging is not complicated by dependent types. The staged interpreter `evalS`, returns a value of type ◯`(typEval t)`. Note that the type of value assignments is also changed (see `envEvalS` in Figure 2.6): Rather than carrying runtime values for L_0, it carries pieces of code representing the values in the variable assignment:

```
fun envEvalS (e : Env) : *1 =
 case e of EmptyE → unit | ExtE e2 t → (envEvalS e2, ◯(typEval t))
```

```
fun evalS (e : Env) (rho: envEvalS e) (s : Exp) (t : Typ)
          (j : J(e,s,t)) : (◯(typEval t)) =
 case j of
    JN e1 n1 → ⟨n1⟩
  | JV e1 t1 → #2(rho)
  | JW e1 t1 t2 i j1 → evalS e1 (#1(rho)) (EV i) t1 j1
  | JL ee1 et1 et2 es2 ej1 →
    ⟨fn v:(typEval et1) → (~(evalS (ExtE ee1 et1) (rho,⟨v⟩) es2 et2 ej1))⟩
  | JA e s1 s2 t1 t2 j1 j2 →
    ⟨~(evalS e rho s1 (ArrowT t1 t2) j1) ~(evalS e rho s2 t1 j2)⟩
```

Figure 2.6: Staged tagless interpreter (without representation types)

Even though the eval function never performs tagging and untagging, the interpretative overhead from traversing its input is still considerable. Proofs of judgments must be deconstructed by eval at run-time. This may require even more work than deconstructing tagged values. With staging, all these overheads are performed in the first stage, and an overhead-free term is generated for execution in a later stage. Executing the function `evalS` produces the tagless code fragments that we are interested in. For example, if we construct and then evaluate the typing judgment for the expression `(EA (EL NatT (EV 0)) (EI 1))`, the code generated by `evalS` looks something like this: ⟨`(fn (x : Nat) => x) 1`⟩.

Staging violations are prevented in a standard way by Meta-D's type system. The staging constructs are those of Davies [30] with the addition of cross-stage persistence [135]. We refer the reader to these references for further details on the nature of staging violations. Adding a run construct along the lines of previous works [130, 82, 134] was not considered here.

Now we turn to addressing some practical questions that are unique to the dependent typing setting, including how the above-mentioned judgments are constructed.

2.5 Constructing Proofs of Typing Judgments

Requiring the user of a L_0 interpreter to construct and supply the proof of a typing judgment for each program to be interpreted is not likely to be acceptable (although it can depend on the situation). The user should be able to use the implementation by supplying only the plain text of the object program. Therefore,

```
1  fun tcVar (e:Env) (n:Nat) : ([t:Typ](JV (e,n,t))) =
2  case n of
3     Z→ (case e of ExtE e' t' → [t=t'](VZ e' t'))
4   | S n'→
5       (case e of ExtE e' t2 →
6          case (tcVar e' n') of
7              [rx:Typ]j2 →[t=rx](JW e2 rx t2 n' j2))
8
9  fun typeCheck (e : Env) (s : Exp)  : ([t : Typ] J(e,s,t)) =
10 case s of
11    EI n→[t = NatT](JN e n)
12  | EV idx→
13     let [rt:Typ]jv = tcVar e idx
14     in  [t=rt](JV e idx rt jv)
15  | EL targ s2→
16     let [rt:Typ]j = (typeCheck (ExtE e targ) s2)
17     in  [t=ArrowT targ rt](JL e targ rt s2 j)
18  | EA s1 s2→
19      let  [rt1:Typ]j1 = (typeCheck e s1)
20           [rt2:Typ]j2 = (typeCheck e s2)
21      in case rt1 of
22           ArrowT tdom tcod→
23             [t=tcod](JA e s1 s2 tdom tcod j1
24                         (cast [assert rt2=tdom, fn (t:Typ)→J(e,s,t), j2]))
```

Figure 2.7: The function `typeCheck` (without representation types)

the implementation needs to include at least a type checking function. This function takes a representation of a type-annotated program and constructs the proof of the appropriate typing judgment, if it exists. We might even want to implement type inference, which does not require type annotations on the input. Figure 2.7 presents a function `typeCheck`. This function is useful for illustrating a number of features of Meta-D:

Dependent sums. The type of the result[4] of `typeCheck` is a dependent sum, written `[t:Typ] J(e,s,t)`[5]. This means that the result of `typeCheck` consists of an L_0 type, and a typing judgment that proves that the argument expression has that particular type under a given type assignment.

Since proofs of judgments are built from sub-proofs of sub-expression judgments, a `case` construct in(strong dependent sum elimination) is need to deconstruct the results of recursive calls to `typeCheck`.

Equality types. The case for constructing proofs of application judgments (Figure 2.7, lines 18–24) illustrates an interesting point. Building a proof for the judgment of the expression `(EA s1 s2)` first involves computing the proofs for the sub-terms `s1` and `s2`. These judgments assign L_0 types `(ArrowT tdom`

[4] In a pure setting (that is with no computational effects whatsoever) the result of `typeCheck` should be `option ([t : Typ] (J (e,s,t)))`, since a particular term given to `typeCheck` may not be well-typed. In the function given in this paper, we omit the `option`, to save on space (and rely on incomplete case expressions instead).

[5] A note on the notation: the dependent product types $\Pi x : \tau_1.\tau_2$ are written as `(x:t1)→t2` in MetaD. Analogously, we shall write dependent sum types using similar notation, replacing the parentheses with angle brackets. Thus, $\Sigma x : \tau_1.\tau_2$ is written as `[x:t1]t2`.

`tcod)` and `rt2` to expressions `s1` and `s2`, respectively.

However, by definition of the inductive family `J`, in order to build the proof of the larger application judgment, `tdom` and `rt2` must be the same L_0 type. The equality between `tdom` and `rt2` must be known statically, at type checking time of the function `typeCheck` so that the L_0 judgment for the application can be constructed. In particular, we must "cast", for example, from the type `(J(e, s2,rt2))` to `(J(e, s2,tdom))`.

How can we do this? A standard way, in type theory, to deal with problems like this is to introduce a type family representing equality over particular values. Such a type family may look something like this in MetaD :

```
inductive EQ (a:*1, x : a) : a→*1 =
 EQ_Refl : (EQ x x)
```

Next, we define a function that can perform *substitution of equals for equals:*

```
fun eqForEq : (a:*1) (x,y : a) (EQ a x y) (f : a -> *1) (f x) : (f y) = ...
```

The function `eqForEq` takes a proof that two values of type `a`, `x` and `y`, are equal. The next argument, `f`, is a function describing a type in terms of a *value* of type `a`. Next, a value of type `(f x)` is taken, and (since `x` and `y` are equal) returns a value of type `(f y)`.

One question remaining is how to construct the proof `EQ a x y`? This cannot be answered in general, but for particular inductive types such as the data-type `Typ`, representing L_0 types, such proofs can be constructed by inductively examining two terms, and combining proofs of equalities of sub-terms to produce proofs of equalities of larger terms:

```
fun isEqTyp : (x : Typ) (y : Typ) : (option (EQ Typ x y)) =
 case x of
     NatT → (case y of NatT → (SOME (EQ_Refl NatT))
                     | x    → NONE)
   | (ArrowT t1 t2) →
       (case y of ArrowT t3 t4 →
                (case (isEqTyp t1 t3, isEqTyp t2 t4) of
                      (SOME p1, SOME p2) → ...
                    | _ → NONE)
              | x → NONE)
```

Assert/cast. In our presentation of Meta-D, we shall examine an alternative to the style of equality described above. We add two language constructs to Meta-D to express this sort of constraint between values. First, the expression of the form `assert` e_1=e_2 introduces an *equality judgment*, `(EQ t` e_1 e_2`)`,

between values of equality types. The type `EQ t` here is *not* the inductive family `EQ` defined above, although it is designed to perform a similar role. Instead, it is treated as a primitive type, whose introduction construct are the `assert` expressions.

An elimination construct `cast[e1,T,e2]` is introduced to perform casting based on an asserted equality. The typing rule for `cast` is as follows:

$$\frac{\Gamma \vdash e_1 : \mathsf{EQ}\ \tau\ t_1\ t_2 \quad \Gamma \vdash T : \tau \to *1 \quad \Gamma \vdash e_2 : (T\ t_1)}{\Gamma \vdash \mathsf{cast}[e_1, T, e_2] : (T\ t_2)}\text{CAST}$$

The `cast` expression takes three arguments: the first is the proof of equality between two values t_1 and t_2 of type τ; the second is a function T that compute a type $*1$ dependent on a τ value. Finally, it takes an expression of $(T\ t_1)$ and converts it to an expression of type $(T\ t_2)$.

Operationally, the expression `assert e1=e2` evaluates its two subexpressions and compares them for equality. If they are indeed equal, computation proceeds. If, however, the two values are not equal, the program raises an exception and terminates. Note that this forces us to use `assert` only over types of values that *can* be compared for equality at runtime. This would include integers, strings, various (ground) datatypes, but exclude functions, along the lines of automatically derived *equality types* in Standard ML [80].

The `cast` construct makes sure that its equality judgment introduced by assert is strictly evaluated (resulting either in an equality proof or in runtime error), and if the equality check succeeds, acts simply as an identity on the second argument `e2`.

The `assert/cast` is intended primarily to serve as a convenient programming shortcut and relieve the user from the effort of explicitly constructing equality proofs. It has no analog in type theory. The programmer need not use it: one can always use the `EQ`-like encoding of equality and construct equality proofs by examining the terms involved inductively.

We examine the function `typeCheck` in some detail:

```
fun typeCheck (e : Env) (s : Exp)  : ([t : Typ] J(e,s,t)) =
case s of
  . . .      . . .      . . .
```

1. **Constant case.** We start with an integer constant expression `EI n`. We know that the resulting L_0 judgment has the (L_0) type `NatT`. Thus, we build a dependent sum "package," `[t=NatT](JN e n)`, which has the (Meta-D) type `[t:Typ](J(e,s,t))`:

   ```
   fun typeCheck (e:Env) (s:Exp) : ([t:Typ]J(e,s,t)) =
   case s of
      EI n→[t = NatT](JN e n)
   ```

2. **Variable case.** Following the usual pattern, we will use an auxiliary function `tcVar` to construct the proof a variable judgment, which can then be plugged into the proof for the variable-expression judgment.

```
fun tcVar (e:Env) (n:Nat) : ([t:Typ](JV (e,n,t))) =
case n of
   Z→(case e of ExtE e' t' → [t=t'](VZ e' t'))
 | S n'→ (case e of ExtE e' t2 →
             case (tcVar e' n') of
                 [rx:Typ]j2 →[t=rx](JW e2 rx t2 n' j2))
```

3. **Abstraction case.** The expression `s` is of the form `(EL targ s2)`, where the L_0 type `targ` is the type of the function's argument, and the L_0 expression `s2` is the body of the λ-abstraction. Type checking proceeds by first extending the type assignment `e` with the L_0 type of the function's argument, and computing the proof for the abstraction body `s2` in the extended type assignment.

   The recursive call to `typeCheck` returns a dependent sum `[rt:Typ]j`. The variable `rt` is bound to the L_0 type of the abstraction expression's body. The variable `j`, which has the (Meta-D) type `J(Ext e targ, s2, targ)`, is bound to the corresponding proof of the typing judgment for the abstraction body computed by the recursive call to `typeCheck`. Finally, the type for the λ-abstraction is returned as `(ArrowT targ rt)`, and combined with the abstraction judgment proof `(JL e targ rt s2 j)`:

```
fun typeCheck (e : Env) (s : Exp)  : ([t : Typ] J(e,s,t)) =
case s of
    EL targ s2→
      let [rt:Typ]j = (typeCheck (ExtE e targ) s2)
      in [t=ArrowT targ rt](JL e targ rt s2 j)
```

4. **Application case.** Starting with the L_0 application `(EA` s_1 s_2`)`, we first compute the judgment proof and type for each of the sub-expressions s_1 and s_2. Next, we check that the (L_0) type index rt_1 computed for the expression s_1 is indeed an arrow type with domain `tdom` and codomain `tcod`.

   In order to build proof of the typing judgment for the entire application expression, we must ensure that the type index of the judgment for the argument expression s_2 must be equal to `tdom`. To this end, we use `cast` to convert the judgment (j_2 : `J(e,`s_2`,`rt_2`)`) to `J(e,`s_2`,tdom)` which is the type we need to construct the proof of the judgment for the entire application expression.

```
fun typeCheck (e : Env) (s : Exp)  : ([t : Typ] J(e,s,t)) =
case s of
    EA s1 s2 →
      let  [rt1:Typ]j1 = (typeCheck e s1)
           [rt2:Typ]j2 = (typeCheck e s2)
      in case rt1 of
           ArrowT tdom tcod→
             [t=tcod](JA e s1 s2 tdom tcod j1
                      (cast [assert rt2=tdom, fn (t:Typ)→J(e,s,t), j2]))
```

2.6 Representation Types

Another practical concern is that types that depend on values can lead to either undecidable or unsound type checking in the meta-language. This happens when values contain diverging or side-effecting computations. In this section we discuss how both of these concerns can be addressed in the context of Meta-D. Combining effects with dependent types requires care. For example, the `typeCheck` function is partial, because there are many input terms which are just not well typed in L_0. Such inputs to `typeCheck` would cause runtime pattern match failures, or an equality assertion exception. We would like Meta-D to continue to have side-effects such as non-termination and exceptions. At the same time, dependently typed languages perform computations during type checking (to determine the equality of types). If we allow effectful computations to leak into the computations that are done during type checking, then we risk non-termination, or even unsoundness, at type-checking time. Furthermore, it is in general desirable to preserve the notion of *phase distinction* between compile time and runtime [17], where static (type-checking) computation and dynamic computation (program execution) are as clearly separated as possible.

The basic approach we adopt to dealing with this problem is to allow types to only depend on other types, and not values. But, disallowing all dependencies of types on values would not allow us to express any of the evaluation or type checking functions for the implementation of L_0, since all of their types depend to some degree on the value of its argument.

A standard solution to restoring some of the expressiveness of dependent types is to introduce a mechanism that allows only a limited kind of dependency between values and types. This limited dependency uses so-called singleton or representation types [58, 148, 25, 26, 140]. The basic idea is to allow types to depend not on arbitrary expressions, but rather, just the values of runtime computations. This is achieved by a two-fold mechanism:

1. The language of types and kinds is sufficiently enriched to allow for defining a representation of values at type level: the type language becomes in effect a powerful, but *pure* dependently typed language.

   The idea is that this type language contains not only *types* of runtime values, but also a logic that can be used to describe their properties. This is done by the standard "propositions-as-types" idea, except that everything is lifted one level up: properties of types are represented as (inductive) *kinds*, while proofs of those properties are lifted to the level of types. A special (inductive) kind is reserved to represent types that classify runtime expressions.

2. A runtime, or *computational* language is introduced "below" the pure type language [116]. More importantly, values in the computational language are typed uniquely by their counterparts in the

type language. In MetaD, we shall use the built-in type R to write such singleton types.

For example, the type of the runtime value 1 is written as the *type* (R 1), where 1 is a *type* of *kind* Int. The value 1 in the runtime language is the only member of such a type (hence the name singleton). When types are given to the functions at the computational level, their behavior must be modeled at type level as well.

For example, the runtime function that adds 1 to an integer has the following type:

```
addOne : (n : Nat)(R n) -> (R (succ n))
```

2.6.1 Working with Representation Types

Now, we can rewrite our (pre-MetaD) interpreter so that its type does not depend on runtime values, which may introduce effects into the type-checking phase. Any computation in the type checking phase can now be guaranteed to be completely effect-free. The run-time values are now forced to have representation types that reflect, in the world of values, the values of inductive kinds.

Meta-D provides the programmer with the interface to representation types through two main mechanisms:

1. A special type construct R is used to express representation type dependency.

   For example, we can define an inductive *kind* Nat

   inductive Nat : *2 = **Z** : Nat | **S** : Nat → Nat

   Note that this definition is exactly the same as the one we had for the type Nat, except it is now classified by *2 instead of *1. Elements of Nat are now *types* Z, (S Z), (S (S Z)), and so on.

   The type construct R takes an element of an inductively defined kind such as Nat, and forms a type R(S Z) : *1. The type R (S Z) refers to a type that has a unique inhabitant that is the runtime representation of the number 1.

2. We write the unique value inhabiting the type (R (S Z)) as (rep (S Z)). In other words: (rep (S Z)) : R (S Z).

   If one is to be able to analyze, at runtime, the elements of a representation type R n, an elimination construct is required. In particular, this is done by a form of case analysis on types [55, 25, 147, 114, 26]:

   tycase x by y of C_n x_n → e_n

A pattern `(Cn xn)` matches against a value `x` of type `K`, where `K` is some inductive kind, only if we have provided a representation value `y` of type `R(x)`.

A pattern `(Cn xn)` matches against a representation type `x`, of inductive *kind* `K`. However, since we cannot allow computation at runtime to depend on types (which are available statically), we must also supply a runtime representation of the type `x` (i.e., a value of type `R(x)`.

Inside the body of the case (`en`), the expression `rep xn` provides a representation value for the part of the inductive constructor that `xn` is bound to.

Let us consider a simple example. We well define inductively an addition function that adds two (singleton) naturals together. First note, however, that in order to give a type to this function, we must produce an addition function *at the level of types* (the function `plus'`). This is done using primitive recursion or, as in the example below, a special syntactic sugar for catamorphisms[6]:

```
plus' (m:Nat) (n:Nat) : Nat  =
  cata m : Nat of
      Z →n
    | S m' →S m'

fun plus : (m:Nat) (m':R(m)) (n:Nat) (n':R(n)) :  R(plus' m n) =
 tycase m by m' of
      Z →n'
    | S p→ (rep(S)) (plus p (rep p) n n')
```

2.6.2 Tagless Interpreter with Representation Types

Figure 2.8 presents the implementation with representation types. Introducing this restriction on the type system requires us to turn the definition of `Exp`, `Env`, and `Typ` into definitions of kinds (again this is just a change of one character in each definition):

```
1 type nat = [n:Nat](R(n))
2 inductive Nat : *2  = Z    : Nat | S : (Nat →Nat)

4 inductive Typ : *2 = ArrowT : Typ →Typ →Typ  | NatT   : Typ

6 inductive Exp : *2  = EI : Nat →Exp | EV : Nat →Exp
7                     | EL : Typ →Exp →Exp  | EA : Exp →Exp →Exp

9 inductive Env : *2  = EmptyE  : Env  | ExtE : Env →Typ →Env
```

[6] A more general primitive recursion scheme can be implemented as in, for example, Coq [6]

Because these terms are now kinds, we cannot use general recursion in defining their interpretation. Therefore, we use special primitive recursion (and catamorphism) constructs provided by the type language to define these interpretations:[7]

```
fun typEval (t:Typ)→*1 =
  cata t:Typ of
    NatT→nat
  | (ArrT t1 t2)→(t1→t2)

fun envEval (e:Env) : *1 =
  cata e:Env of
   EmptyE→unit
   ExtE et t→(et,t)
```

Judgments, however, remain a type, of kind `*1`. The reason for this is that typing judgments are used *at runtime* by the interpreter. It is important to note, however, that now judgments are a type indexed by other types, not a dependent family indexed by values.

For the most part, the definition of judgments and the interpretation function do not change. We need to change judgments in the case of natural numbers by augmenting them with a representation for the value of that number. The constructor `JN` now becomes

```
JN : (e1 : Env)→(n  : Nat)→(R n)→ J(e1,EI n,NatT)
```

and the definition of `eval` is changed accordingly. First, we define an auxiliary function `mknat` which converts a `R(n)` for some `Nat n` into the type `nat` which corresponds to the type to which object-language integer expressions are mapped. This function is then used to construct an appropriate value for the **`JN`** case:

```
fun mknat (n : Nat)  (rn : R(n))  : nat =
   tycase n by rn of
       Z→[n=zero](rep zero)
     | S n2→
         case (mknat n2 (rep n2)) of
           [n2':Nat]rn2'→ [n=(S n2')](rep(S) rn2')

fun eval (e : Env) (rho: envEval e) (s : Exp) (t : Typ)
         (j : J(e,s,t)) : (typEval t) =
```

[7] These constructs are similar to primitive recursive schemata that the Coq theorem prover derives for inductively defined type families – this technique can be readily reused in Meta-D. Alternatively, the functions can be defined using recursion, and a termination check (as, for example, in Alfa [52]) conducted before the functions are admitted by the system. The latter is currently the case, although our implementation of the termination check is, at this time, based on a rather simple syntactic criterion.

```
29  case j of
30   JN  e1 n1 rn1→mknat n1 rn1
31    . . .     . . .    . . .
```

Note that even though modified `eval` uses a helper function (`mknat`) to convert a representation of a natural type to a natural number, in practice, we see no fundamental reason to distinguish the two. Identifying them, however, requires the addition of some special support for syntactic sugar for this particular representation type.

The remainder of the function `eval`, together with other parts of the implementation using representation types is given in Figure 2.8. It may be surprising to note that other than the changes mentioned above, there are no further modification to the text of the programs that needs to be made to the ones presented in the pure non-representation type setting.

2.6.3 `typeCheck` with Representation Types

The full definition of `typeCheck` is given at the bottom of Figure 2.8.

Let us first examine the type signature of the new version of `typeCheck`.

```
1  fun typeCheck (e : Env) (re: R(e))
2                (s : Exp) (rs: R(s))  : ([t : Typ] (R(t),J(e,s,t))) = . . .
```

Three things are worth noting:

1. The function still returns a sum result consisting of an object language type and a proof of the judgment that the argument expression has that type. However, because `Typ` has been promoted to an inductive *kind*, the sum returned is more like an existential type than a dependently typed strong sum. In Meta-D notation, both are written the same way.

2. Note also, that the result, in addition to the proof of the judgment, contains a runtime representation of the object-language type, `R(t)`, where `t` is the resulting object-language type. This is necessary in order to compare the object-language types returned by different recursive invocations of `typeCheck` since the `tycase` construct requires both a `t` and a `R(t)` to compare types at runtime.

3. Similarly, the arguments to `typeCheck` are not only `Env`s and `Exp`s , but their respective representations. Again, this is necessary because of the `tycase` construct cannot examine the structure of the argument expressions or type assignments without their runtime representation.

2.7 Conclusion

In this chapter we have shown how a dependently typed programming language can be used to express a staged interpreter that completely circumvents the need for runtime tagging and untagging operations associated with universal datatypes. In doing so we have highlighted two key practical issues that arise when trying to develop staged interpreters in a dependently typed language. First, the need for functions that construct the proofs of typing judgments that the interpretation function *should* be defined over. And second, the need for representation types to avoid polluting the type language with the impure terms of the computational language.

To demonstrate that staging constructs and dependent types can be safely combined, in the next chapter we shall formally develop a multi-stage computational language typed by Shao, Saha, Trifonov, and Papaspyrou's TL system [116]. This allows us to prove type safety in a fairly straightforward manner, and without having to duplicate the work done for the TL system.

A practical concern about using dependent types for writing interpreters is that such systems do not have decidable type *inference*, which some view as a highly-valued feature for any typed language.

In terms of programming, we have first started with a *Coq* implementation of a tagless interpreter. Next, we explored a dependently type programming language. We were guided by the idea of designing the meta-language that would be more accessible to a programmer than to a logician. We did not find that the explicit type annotations and new constructs were an excessive burden, and some simple tricks in the implementation of the meta-language could be enough to avoid the need for many such redundant annotations. However, representation types do seem to complicate our programs somewhat.

In later chapters, we shall explore how much of the style of the tagless interpreter implementation could be implemented in a more main-stream setting of Haskell.

```
inductive nat : *1  = zero : nat | succ : (nat→nat)
inductive Nat : *2  = Z    : Nat | S : (Nat→Nat)

inductive Typ : *2 = ArrowT : Typ→Typ→Typ  | NatT   : Typ

inductive Exp : *2  = EI : Nat→Exp | EV : Nat→Exp
                    | EL : Typ→Exp→Exp  | EA : Exp→Exp→Exp

inductive Env : *2  = EmptyE  : Env  | ExtE : Env→Typ→Env

inductive J : (Env, Exp, Typ)→*1  =
  JN :(e1 : Env)→(n:Nat)→(rn : R n)→J(e1,EI n,NatT)
| JV :(e1 : Env)→(t1:Typ)→J(ExtE e1 t1,EV Z,t1)
| JW :(e1 : Env)→(t1 : Typ)→(t2 : Typ)→(i : Nat) →
      J(e1,EV i,t1)→J(ExtE e1 t2,EV (S i), t1)
| JL :(e1 : Env)→(t1 : Typ)→(t2 : Typ)→(s2 : Exp)→
      J(ExtE e1 t1,s2,t2)→J(e1,EL t1 s2,ArrowT t1 t2)
| JA :(e : Env)→(s1 : Exp)→(s2 : Exp)→(t1 : Typ)→(t2 : Typ)→
      J(e,s1,ArrowT t1 t2)→J(e,s2,t1)→J(e,EA s1 s2,t2)


val typEval : Typ → *1 =
    cata Typ nat (fn c : *1→fn  d : *1→(c→d))

val envEval : Env → *1 =
   cata Env unit (fn r : *1→fn t : *1→(r,t))

fun cast (n : Nat)  (rn : R(n))  : nat = tycase n by rn of Z→zero
                                                 | S n2 →succ (cast n2 (rep n2))

fun eval (e : Env) (rho: envEval e) (s : Exp) (t : Typ) (j : J(e,s,t)) : (typEval t) =
 case j of
  JN  e1 n1 rn1→cast n1 rn1
| JV  e1 t1 →#2(rho)
| JW  e1 t1 t2 i j1→eval e1 (#1(rho)) (EV i) t1 j1
| JL ee1 et1 et2 es2 ej1→fn v:(typEval et1)→(eval (ExtE ee1 et1) (rho,v) es2 et2 ej1)
| JA e s1 s2 t1 t2 j1 j2→(eval e rho s1 (ArrowT t1 t2) j1) (eval e rho s2 t1 j2)

fun typeCheck (e : Env) (re: R(e)) (s : Exp)  (rs: R(s))  : ([t : Typ] (R(t),J(e,s,t))) =
tycase s by rs of
  EI n→[t = NatT] (NatT',(JN e n (rep n)))
| EV n→
 (tycase n by (rep n) of Z→(tycase e by re of ExtE ee  t2→[t = t2](rep t2, JV ee t2))
                      | S n→(tycase e by re of ExtE (e2) (t2)→
                                  ((fn x : ([t:Typ] (R(t), J(e2,EV n,t)))→
                                     case x of [rx : Typ]j2→([t = rx]
                                                             (#1 j2, JW e2 rx t2 n (#2 j2)))
                                  (typeCheck e2 (rep e2) (EV n) (rep (EV n)))))))
| EL targ s2→
 ((fn x : ([t : Typ](R(t),(J(ExtE e targ,s2,t)))) =>
     case x of [t : Typ] j2→
      [t = ArrowT targ t] (rep (ArrowT targ (#1 t))), (JL e targ t s2 (#2 j2)) )
 (typeCheck (ExtE e targ) (rep (ExtE e targ)) s2 (rep s2)))
| EA s1 s2→
 ((fn x1 : [t1 : Typ](R(t1),(J(e,s1,t1)))→(fn x2 : [t2 : Typ](R(t2),(J(e,s2,t2)))→
    case x1 of [t1 : Typ]j1→case x2 of [t2 : Typ]j2→
     (tycase t1 by (#1 (j1)) of
       ArrowT tdom tcod→
        [t = tcod] (rep tcod, (JA e s1 s2 tdom tcod j1
                                  (cast [assert t2=tdom,J(e,s,tdom),j2])))) end)))
 (typeCheck e (rep e) s1 (rep s1)) (typeCheck e (rep e) s2 (rep s2)))
```

Figure 2.8: Tagless interpreter with representation types in MetaD

Chapter 3

Staging and Dependent Types: Technical Results

3.1 Introduction

This chapter[1] is intended as a technical prolegomenon to the exploration of meta-theoretic properties of the meta-language MetaD used in the previous chapter. In particular, we are concerned with type safety properties meta-languages such as the language Meta-D. The result we report here is type safety for a formalized core subset of Meta-D. This result shows that multi-stage programming constructs can be safely used, even when integrated with a sophisticated dependent type system such as that of TL [116].

Let us first explain what is meant by "a formalized core subset of Meta-D." Formalizing a rather large programming language in which our examples in Chapter 2 have been written seems somewhat impractical: many details would overwhelm our ability to (a) manipulate the formal constructs effectively; and (b) clearly demonstrate the most essential features that we are trying to study. Thus, we shall cut down the formalism to its bare essentials, illustrating the following points:

1. The meta-language we present includes singleton (representation) types. Instead of general inductive family definitions, the language has a couple of "built-in" singleton types such as natural numbers and booleans. Later, we shall expound on how the formal treatment can be extended to more complex data-types.

2. The meta-language is designed to support *staging* with code-brackets and escape. With this we intend to show that staging can safely interact with other features under consideration.

3. We shall formalize the `assert/cast` expressions used in Chapter 2.6 and show that they, too, can be integrated into a meta-language in a type-safe way.

[1] This chapter is based on previously published material [102, 103].

Another interesting feature of the presentation is the use of the TL [116] framework in our formal development. Essentially, the idea is to obtain a powerful type system for the programming language by simply reusing a general theoretical framework for such languages developed by Shao *et al* as a part of the FLINT project. This allows us to reuse many of their technical results without having to prove them from scratch.

3.2 The Language $\lambda_{H\bigcirc}$

In this section we will define and discuss the language $\lambda_{H\bigcirc}$[2] which is a formalization of the ideas described above. First, we review some of the properties and definition of the TL framework which is used to define $\lambda_{H\bigcirc}$. Second, we define the syntax and static semantics of the language. Third, we define a small-step semantics of $\lambda_{H\bigcirc}$, and, finally, prove the type safety of $\lambda_{H\bigcirc}$.

3.2.1 Review: TL

TL is a general framework intended for designing sophisticated typed intermediate languages for compilers [116]. The basic architecture of the system is as follows:

1. Different *computational languages* can be defined. These are typed programming languages or calculi intended for writing executable programs. As such, they can have effects such as non-termination, state and so on. However, the types for these computational languages are provided by the common *type language* TL. Several computational languages are presented by Shao *et al*. In this chapter we implement our own computational language with the features enumerated above.

2. The type language TL is a typed specification language in the spirit of the Calculus of Inductive Constructions. This language supports dependent types, higher-order kinds and inductive families. It is intended for two purposes:

   (a) To describe the behaviors of computational/runtime programs in a pure, logical way, to represent logical properties of these programs and encode the proofs of these properties in a type-theoretic way.

   (b) A set of the computational language types is defined as an inductive kind in TL. Many different computational languages share TL as their type/specification language. The advantages of this are again twofold:

[2] A brief note on the name $\lambda_{H\bigcirc}$: $\lambda_{H\bigcirc}$ (pronounced "lambda H-circle") is derived from the name of the calculus λ_H, of uncertain provenance defined by Shao *et al.* [116] The circle has been appended to the name to indicate the addition of staging constructs, similar to Davies' naming of the calculus $\lambda^{\bigcirc}$[29]).

i. Many computational languages can be put together into a single meta-theoretical framework, where translation between them can be expressed and studied. In particular transformations from one language into another can be written in a way that, in some clearly defined sense, preserves (or transforms) types between them [116].

ii. TL promotes reuse in defining and proving properties about computational language, since meta-theoretical properties of TL are established once and for all. Such properties include "subject reduction, strong normalization, Church-Rosser (and confluence), and consistency of the underlying logic" [116].

The definitions and basic properties of TL that we reuse here are available in the Shao *et al* technical report [115]. Using the TL framework, we can arrive at an advantageous division of labor. In this chapter, we formally define and prove properties of a new computational language (1), while most theoretical work for the type language (2) can be simply reused from existing literature.

3.2.2 The language $\lambda_{H\bigcirc}$

We follow the same approach used by the developers of TL, and build a computation language $\lambda_{H\bigcirc}$ that uses TL as its type language. Integrating our formalization into the TL framework gives us significant practical advantages in formal development of $\lambda_{H\bigcirc}$:

1. Important meta-theoretic properties of the type language we use, TL, have already been proved [116]. Since we do not change anything about the type language itself, all these results (e.g., the Church-Rosser property of the type language, decidable equality on type terms) are easily reused in our proofs.

2. $\lambda_{H\bigcirc}$ is based on the computational language λ_H [116]. We have tried to make the difference between these two languages as small as possible (essentially, just the addition of staging constructs). As a result, the proof of type safety of $\lambda_{H\bigcirc}$ is very similar to the type safety proof for λ_H. Again, we were able to reuse certain lemmata and techniques developed by Shao and others for λ_H in our own proof.

The Syntax and Static Semantics of $\lambda_{H\bigcirc}$

Figure 3.1 defines $\lambda_{H\bigcirc}$ computational types, and is the first step needed to integrate $\lambda_{H\bigcirc}$ into the TL framework. The set of types for the computational language is simply the inductive TL *kind* $\Omega^{\bigcirc}$, which is comprised of the following:

inductive Nat : Kind	::=	$0, 1, 2, \ldots$	
inductive Bool : Kind	::=	true | false	
inductive $\Omega^{\bigcirc}$: Kind	::=	snat	: Nat $\rightarrow \Omega^{\bigcirc}$
		| sbool	: Bool $\rightarrow \Omega^{\bigcirc}$
		$\mid \twoheadrightarrow$	: $\Omega^{\bigcirc} \rightarrow \Omega^{\bigcirc} \rightarrow \Omega^{\bigcirc}$
		| tup	: Nat $\rightarrow$ (Nat $\rightarrow \Omega^{\bigcirc}$) $\rightarrow \Omega^{\bigcirc}$
		$\mid \forall_k$	: Πk : Kind.$(k \rightarrow \Omega^{\bigcirc}) \rightarrow \Omega^{\bigcirc}$
		$\mid \exists_k$	: Πk : Kind.$(k \rightarrow \Omega^{\bigcirc}) \rightarrow \Omega^{\bigcirc}$
		$\mid \forall_{KS}$	: Πk : KScheme.$(k \rightarrow \Omega^{\bigcirc}) \rightarrow \Omega^{\bigcirc}$
		$\mid \exists_{KS}$	: Πk : KScheme.$(k \rightarrow \Omega^{\bigcirc}) \rightarrow \Omega^{\bigcirc}$
		$\mid \bigcirc$	: $\Omega^{\bigcirc} \rightarrow \Omega^{\bigcirc}$
		| EQ	: Nat $\rightarrow$ Nat $\rightarrow \Omega^{\bigcirc}$

Natural number operators

$\oplus$: Nat $\rightarrow$ Nat $\rightarrow$ Nat
$\oplus \in \{+, -, \times, \ldots\}$
$\circledcirc$: Nat $\rightarrow$ Nat $\rightarrow$ Bool
$\circledcirc \in \{\leq, \geq, =, \ldots\}$

Figure 3.1: The TL definition of the types of $\lambda_{H\bigcirc}$

1. *Singleton types* sbool *and* sint*.* These types illustrate the central concept in the type system of $\lambda_{H\bigcirc}$. They take an argument of the TL inductive kind Nat (or Bool) a $\lambda_{H\bigcirc}$ type (of the inductive kind $\Omega^{\bigcirc}$) that classifies *individual natural number (or boolean) values* in the computational language.

   We shall write a hat $\hat{\cdot}$ over natural number literals in the computational language where necessary to disambiguate between them and TL natural numbers of kind Nat. For example, an integer constant $1\hat{0}\hat{0}$ in the computational language has the type (snat 100). Similarly, the $\lambda_{H\bigcirc}$ expression $\hat{1}\hat{+}\hat{1}$ has the type (snat $1 + 1$) which is equal (in TL) to the type (snat 2).

2. *Arrow types.* These are simply the types of functions in $\lambda_{H\bigcirc}$. For example, the computational-language addition operator has the following arrow type for any m, n : Nat:

$$\hat{+} : (\text{snat } m) \twoheadrightarrow (\text{snat } n) \twoheadrightarrow (\text{snat } (m + n))$$

3. *Tuple types.* Tuple types are represented by two pieces of information: first, the natural number argument representing the size of the tuple, and, second, a function that, given a natural number i, returns the $\lambda_{H\bigcirc}$ type of the i-th component of the tuple. For example, the pair type $A \times B$ would be represented by

$$(\text{tup } 2\ (\lambda n.\text{case } n \text{ of } 0 \rightarrow A \mid 1 \rightarrow B \mid \_ \rightarrow \forall\alpha.\alpha))$$

.

4. *Universal and existential types.* Universal and existential types are essential in $\lambda_{H\bigcirc}$. They can quantify over TL inductive kinds, such as nat, including the kind $\Omega^{\bigcirc}$ itself.

   We use the syntactic following syntactic sugar for writing universals and existentials , omitting the sort s where it is clear from the context:

   $$\begin{aligned} \forall_s\, X : A.\, B &\equiv \forall_s\, A\ (\lambda X : A.\, B) \\ \exists_s\, X : A.B &\equiv \exists_s\, A\ (\lambda X : A.B) \end{aligned}$$

   A brief note on terminology: the sort Kind is a (in TL terminology) *kind schema*, which classifies kinds. The sort $\mathsf{KScheme}$ is a singleton sort that classifies kind schemas. The reason for this terminology is that in TL all levels are lifted up by one: types play the role of programs/proofs, kinds play the role of types/propositions, and kind schemas play the role of kinds.

   The universal quantifier allows us to form types that are polymorphic in *singleton values*, such as the type $(\forall X : \mathsf{Nat}.\, (\mathsf{snat}\ X) \to (\mathsf{snat}\ X))$, which is the type of the identity function over natural number values in $\lambda_{H\bigcirc}$. It also allows us to use a standard notion of polymorphism. For example, the type $(\forall X : \Omega^{\bigcirc}.X \to X)$ is the type of the polymorphic identity function in $\lambda_{H\bigcirc}$.

   Existential quantifiers are very important as well. Recalling that in $\lambda_{H\bigcirc}$ each natural number has a different type, we can use existential types to represent the more usual type of *all natural numbers*. For example, the following definition is such a type of all natural numbers:

   $$\mathsf{CompNat} : \Omega^{\bigcirc} \equiv \exists n : \mathsf{Nat}.\, \mathsf{snat}\, n$$

   . Similarly, we can "lift" the addition operation to work on the $\mathsf{CompNat}$ type as follows:

   $$\begin{aligned} &\mathsf{plus} \quad : \mathsf{CompNat} \multimap \mathsf{CompNat} \multimap \mathsf{CompNat} \\ &\mathsf{plus} \quad = \lambda x : \mathsf{CompNat}.\, \lambda y : \mathsf{CompNat}. \\ &\qquad\qquad \mathsf{open}\ x\ \mathsf{as}\ M, m\ \mathsf{in}\ \mathsf{open}\ y\ \mathsf{as}\ N, n\ \mathsf{in}\ \{r = M + N, m \hat{+} n : (\mathsf{snat}\ r)\} \end{aligned}$$

   The function plus works by first opening its arguments, adding them, and packing them up into a new existential package. Note that two addition operations are used: one at the type level $(+)$ and one and the computational level $(\hat{+})$.

5. *Code type.* Code type is the type of (homogeneous) object program. It is modeled on the circle modality of Davies [29]. Intuitively the type $(\bigcirc(\mathsf{snat}\ 1))$ is the type of computational language program that, when executed, would yield the result 1.

6. *Equality type.* We will also add an equality type over natural numbers to $\Omega^{\bigcirc}$. Intuitively $\mathsf{EQ}\ m\ n$ is a type of proofs that m equals n. We use these types to type the `assert/cast` constructs in the computational language.

Level-indexed syntax. The syntax of the computational language $\lambda_{H\bigcirc}$ is given in Figure 3.2. Instead of a single inductive set of expressions, we give a family of sets of expressions. This family is indexed by a natural number representing the *level* of the expression. A set of values is defined similarly. This technique of presenting syntax of staged language, called *level-indexed families* [29, 128] has become a standard tool for defining reduction (and small-step) semantics of staged languages. Intuitively, the level-indexed families are designed to prevent certain unsafe operations. The family E^0 is defined to exclude top-level escapes, for example. Thus, the reductions such as β are restricted only to E^0 expressions; code operations such as *escapes* can be performed only on E^1 expressions. Without these restrictions, the reduction semantics of staged languages is unsound [129].

The language $\lambda_{H\bigcirc}$ contains recursion and staging constructs. It contains two predefined representation types: naturals and booleans. The `if` construct, as in λ_H [116], provides for propagating proof information into branches (analogous to the `tycase` construct of MetaD); full implementation of inductive datatypes in the style of MetaD is left for future work. Since arbitrary dependent types are prohibited in $\lambda_{H\bigcirc}$, we use universal and existential quantification to express dependencies of values on types and kinds. For example, the identity function on naturals is expressed in $\lambda_{H\bigcirc}$ as follows:

$$(\Lambda n : Nat.\, \lambda x : (\text{snat } n).x) : \forall n : Nat.\, (\text{snat } n) \rightarrow \text{snat } n$$

In $\lambda_{H\bigcirc}$, we also formalize the `assert/cast` construct, which requires extending the language of computational types with equality judgment types. Similarly, we add the appropriate constructs to the syntax of $\lambda_{H\bigcirc}$.

Remark 1 (Level-indexed syntactic families) *1.* $\forall n \in \mathbb{N}. E^n \subseteq E^{n+1}$.

2. $\forall n \in \mathbb{N}. n \geq 1 \Rightarrow V^n = E^{n-1}$.

PROOF. Proof of (1) is constructed easily by induction on the judgment $e \in E^n$. Similarly for (2). □

Typing Judgments. The typing judgment of $\lambda_{H\bigcirc}$ (Figure 3.4) has the form $\Delta; \Gamma \vdash^n e : A$. It has two type assignments

1. $\Delta \in \mathsf{Sequence}\ (X \times \mathbb{N} \times A)$ is a type assignment that maps TL type variables to TL expressions. Also, each mapping carries a natural number indicating the *level* at which the variable is bound. A level-annotation erasure function $(\cdot|_n)$ is used to convert $\lambda_{H\bigcirc}$ typing assignments Δ into a form required by the typing judgment of TL[116]. This interface then allows us to reuse the original TL typing judgment in the definition of the typing judgment for $\lambda_{H\bigcirc}$.

$$
\begin{array}{rcl}
X & \equiv & \text{type variables of TL} \\
A & \equiv & \text{type expressions of TL} \\
W & \equiv & \text{variables of } \lambda_{H\bigcirc} \\
exp^0 \in E^0 & ::= & x \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid f^0 \mid \mathsf{fix}\, x : A.f^0 \mid e_1^0\ e_1^0 \mid e^0[A] \mid \{X = A_1, e^0 : A_2\} \\
 & \mid & \mathsf{open}\ e^0\ \mathsf{as}\ X, x\ \mathsf{in}\ e^0 \mid (e_0^0, \ldots e_{n-1}^0) \mid \mathsf{sel}\,[A](e_1^0, e_2^0) \mid e_1^0 \oplus e_2^0 \\
 & \mid & \mathsf{if}\ [A_1, A_2](e^0, X_1.e_1^0, X_2.e_2^0) \mid \langle e^1 \rangle \\
 & \mid & \mathsf{assert}\ e_1^0 : A_1 = e_2^0 : A_2 \mid \mathsf{cast}\ (e_1^0, A, e_2^0) \\
f^n & ::= & \Lambda X : A.e^n \mid \lambda x : A.e^n \\
exp^{n+} \in E^{n+} & ::= & x \mid m \mid \mathsf{tt} \mid \mathsf{ff} \mid f^{n+} \mid \mathsf{fix}\, x : A.f^{n+1} \mid e^{n+}\ e^{n+} \mid e^{n+}\ [A] \\
 & \mid & \{X = A_1, e^{n+} : A_2\} \mid \mathsf{open}\ e^{n+}\ \mathsf{as}\ X, x\ \mathsf{in}\ e^{n+} \\
 & \mid & (e_0^{n+}, \ldots, e_{m-1}^{n+1}) \mid \mathsf{sel}\,[A](e_1^{n+}, e_2^{n+}) \mid e_1^{n+} \oplus e_2^{n+} \mid \langle e^{n++} \rangle \mid \tilde{}\, e^n \\
 & \mid & \mathsf{if}\ [A_1, A_2](e^{n+}, X_1.e_1^{n+}, X_2.e_2^{n+}) \\
 & \mid & \mathsf{assert}\ e_1^+ : A_1 = e_2^+ : A_2 \mid \mathsf{cast}\ (e_1^+, A, e_2^+) \\
v^0 \in V^0 & ::= & n \mid \mathsf{tt} \mid \mathsf{ff} \mid f^0 \mid \mathsf{fix}\, x : A.f^0 \mid \{X = A_1, v^0 : A_2\} \mid (v_1^0, \ldots, v_{m-1}^0) \mid \langle v^1 \rangle \\
 & \mid & \mathsf{assert}\ v^0 : A = v^0 : B \\
v^1 \in V^1 & ::= & x \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid fv^1 \mid \mathsf{fix}\, x : A.fv_1 \mid v^1\ v^1 \mid v^1[A] \mid \{X = A_1, v^1 : A_2\} \\
 & \mid & \mathsf{open}\ v_1\ \mathsf{as}\ X, x\ \mathsf{in}\ v^1 \mid (v_0^1, \ldots, v_{m-1}^1) \mid \mathsf{sel}\,[A_1](v^1, n^1) \\
 & \mid & v^1 \oplus v^1 \mid \mathsf{if}\ [A_1, A_2](v^1, X_1.v_1^1, X_2.v_2^1) \mid \langle v^2 \rangle \\
v^{n+2} \in V^{n+2} & ::= & x \mid m \mid \mathsf{tt} \mid \mathsf{ff} \mid fv^1 \mid \mathsf{fix}\, x : A.fv_1 \mid v^{n+2}\ v^{n+2} \mid v^{n+1}[A] \mid \{X = A_1, v^{n+2} : A_2\} \\
 & \mid & \mathsf{open}\ v^{n+2}\ \mathsf{as}\ X, x\ \mathsf{in}\ v^{n+2} \mid (v_0^{n+2}, \ldots, v_{m-1}^{n+2}) \mid \mathsf{sel}\,[A](v^{n+2}, n^{n+2}) \\
 & \mid & v^{n+2} \oplus v^{n+2} \mid \mathsf{if}\ [A_1, A_2](v^{n+2}, X_1.v_1^{n+2}, X_2.v_2^{n+2}) \mid \langle v^{n+3} \rangle \mid \tilde{}\, v^{n+1} \\
fv^n & ::= & \lambda x : A.v^n \mid \Lambda X : A.fv^n
\end{array}
$$

Figure 3.2: Stratified syntax of $\lambda_{H\bigcirc}$

2. $\Gamma \in \mathsf{Sequence}\ (W \times \mathbb{N} \times A)$ is a type assignment that maps $\lambda_{H\bigcirc}$variables to their types. Again, each mapping is annotated by the natural number representing the level at which the variable is bound. Intuitively, a type assignment Γ is well-formed (written $\Delta \vdash^n \Gamma$) if for each $(x, n, A) \in \Gamma$, we have $\Delta|_n \vdash A : \Omega^{\bigcirc}$.

The type judgments are indexed by a natural number representing the level at which the typing is performed. When typing an expression surrounded by the code brackets, this number is incremented; similarly, when typing an escaped expression, the number is decremented.

In what follows we shall examine the syntax of $\lambda_{H\bigcirc}$ terms from Figure 3.2. We will introduce each kind of term, and present its typing rule. First, note that there are two sets of object-level variables used in Figures 3.2 and 3.4:

1. A set X type variables. This set ranges over TL types. Individual variables are written as $X, Y, \ldots$. These are basically type variables in the System F and other polymorphic λ-calculi.

2. A set W of $\lambda_{H\bigcirc}$variables, $x, y, \ldots$ that range over $\lambda_{H\bigcirc}$ values.

Also, we will use meta-variable $A, B, \ldots$ to range over type expressions (i.e., expressions of TL).

1. Variable expressions: $e ::= x \mid \cdots$. The typing rule for variables is a rather standard combination of features of λ_H [116] enriched by the MetaML-style level annotations. This particular formulation supports cross-stage persistence by stating that the declaration level m of a variable's use need not be exactly the same as that of its use, n: a variable declared at an earlier stage can be used at a later stage. Changing the $\leq$ into $=$ would give a system without cross-stage persistence similar to that of $\lambda^{\bigcirc}$ [29], and would not fundamentally change our results.

$$\frac{\Delta \vdash^n \Gamma \quad (x : A^m) \in \Gamma \quad m \leq n}{\Delta; \Gamma \vdash^n x : A}\text{Var}$$

2. Constants: $e ::= n \mid \mathsf{tt} \mid \mathsf{ff} \mid \cdots$. The standard natural number constants, as well as Boolean constants are included in the language. Their typing rules are interesting: a Boolean or an integer constant has a sbool (or sint) type directly describing it:

$$\frac{\Delta \vdash^n \Gamma \quad m \in \{0, 1, 2, \cdots\}}{\Delta; \Gamma \vdash^n m : (\mathsf{snat}\ m)}\text{NatConst}$$

$$\frac{\Delta \vdash^n \Gamma}{\Delta; \Gamma \vdash^n \mathsf{tt} : (\mathsf{sbool}\ \mathsf{True})}\text{BoolTrue} \qquad \frac{\Delta \vdash^n \Gamma}{\Delta; \Gamma \vdash^n \mathsf{ff} : (\mathsf{sbool}\ \mathsf{False})}\text{BoolFalse}$$

3. Function and universal abstractions:

$$\begin{aligned} e &::= f \mid e\ e \mid e_1\ e_2 \mid e\ [A] \mid \cdots \\ f &::= \Lambda X : A.e \mid \lambda x : A.e \end{aligned}$$

   Functions bind $\lambda_{H\bigcirc}$variables, while type abstractions bind TL variables. There are, symmetrically, two application forms, one for functions, and the other for type abstractions.

   Unlike simply typed λ-calculus, these rules have some important side conditions:

   - The λ-abstraction rule has a requirement that the explicit type A_1 given to the variable is an $\Omega^{\bigcirc}$type. This is done by invoking the typing judgment of TL: $\Delta|_n \vdash A_1 : \Omega^{\bigcirc}$.
   - In the typing rule for the type abstraction, the type annotation is required to be of one of the *sorts* of TL: $\Delta|_n \vdash B : s$. The same condition is imposed by type application as well.
   - Note that the type variable environment restriction $\Delta|_n$ is used to convert the type environment Δ into a form that the typing judgment of TL can accept.

$$\frac{\Delta|_n \vdash A_1 : \Omega^{\bigcirc} \quad \Delta; \Gamma, x : A_1^n \vdash^n e : A_2}{\Delta; \Gamma \vdash^n (\lambda x : A_1.e) : A_1 \rightarrowtriangle A_2}\text{Abs} \qquad \frac{\Delta; \Gamma \vdash^n e_1 : A_1 \rightarrowtriangle A_2 \quad \Delta; \Gamma \vdash^n e_2 : A_1}{\Delta; \Gamma \vdash^n e_1\ e_2 : A_2}\text{App}$$

$$\frac{\Delta|_n \vdash B : s \quad \Delta, X : B^n; \Gamma \vdash^n f : A}{(\Lambda X : B.\, f) : \forall_s X : B.\, A}\text{TAbs} \qquad \frac{\Delta|_n \vdash A : B \quad \Delta, \Gamma \vdash^n e : \forall_s X : B.A_2}{\Delta; \Gamma \vdash^n e[A] : A_2[X := A]}\text{TApp}$$

4. Existential expressions. Existential types are created using a $(X = A_1, e : A_2)$ expressions with create an object of type $\exists X.A_2$. The corresponding elimination construct is $\mathsf{open}\ e_1\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2$ which deconstructs existential objects.

$$e \quad ::= (X = A_1, e : A_2) \mid \mathsf{open}\ e_1\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2 \mid \cdots$$

Typing rules for existential types are given below. Note that the same side conditions apply as for universal quantification.

$$\frac{\Delta|_n \vdash A_1 : B \quad \Delta|_n \vdash B : s \quad \Delta; \Gamma \vdash^n e : A[X := A_1]}{\Delta; \Gamma \vdash^n (X = A_1, e : A) : \exists_s X : B.A}\text{Pack} \qquad \frac{\Delta, \Gamma \vdash^n e : \exists_s X' : B.A_1 \quad \Delta|_n \vdash A_2 : \Omega^{\bigcirc} \quad \Delta, X : B^n; \Gamma, x : A_1^n[X' := X] \vdash^n e_2 : A_2}{\Delta; \Gamma \vdash^n (\mathsf{open}\ e_1\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2) : A_2}\text{Unpack}$$

5. Fixpoint definitions

$$e \quad ::= \mathsf{fix}\, x : A.f \mid \cdots$$

The fixpoint construct allows for recursive definitions. The body of fix is syntactically restricted to function or type abstraction, since the language is intended to be call-by-value.

$$\frac{\Delta|_n \vdash A : \Omega^{\bigcirc} \quad \Delta; \Gamma, x : A^n \vdash^n f : A}{\Delta; \Gamma \vdash^n (\mathsf{fix}\, x : A.f) : A}\text{Fix}$$

6. Tuples. The tuple formation expression is the fairly standard $(e_0, \ldots, e_n)$.

$$e \quad ::= (e_0, \ldots, e_n) \mid \mathsf{sel}\, [A](e_1, e_2) \mid \cdots$$

$$\frac{0 \leq i < m. \Delta; \Gamma \vdash^n e_i : A_i}{\Delta; \Gamma \vdash^n (e_0, \ldots, e_{m-1}) : \mathsf{tup}\ m\ (\mathsf{nth}\ [A_0, \ldots, A_{m-1}])}\text{Tuple}$$

The elimination construct is a little less standard. It takes three arguments:

1. The type A, which encodes the proof that the index being projected is less than the size of the tuple.

2. The tuple expression itself.

3. The index of the element of the tuple we are trying to project

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash^n e_1 : (\mathsf{tup}\ A_3\ B) \\ \Delta;\Gamma \vdash^n e_2 : \mathsf{snat}\ A_2 \\ \Delta|_n\vdash A : \mathsf{LT}\ A_2 A_3\end{array}}{\Delta;\Gamma \vdash^n \mathsf{sel}\,[A](e_1, e_2) : (B\ A_2)}\text{Select}$$

4. Arithmetical expressions. Assorted arithmetical, comparison and other primitive operators are present wholesale as:

$$e \quad ::= e_1 \oplus e_2 \mid e_1 \circledcirc e_2 \mid \cdots \quad \text{where } \oplus \in \{+, -, *, \ldots\}, \circledcirc \in \{\leq, \geq, =, \ldots\}$$

$$\frac{\begin{array}{l}\oplus : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \Delta;\Gamma \vdash^n e_1 : \mathsf{snat}\ A_1 \\ \Delta;\Gamma \vdash^n e_2 : \mathsf{snat}\ A_2\end{array}}{\Delta;\Gamma \vdash^n (e_1 \hat{\oplus} e_2) : (\mathsf{snat}\ (A_1 \oplus A_2))}\text{Arith} \quad \frac{\begin{array}{l}\circledcirc : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\ \Delta;\Gamma \vdash^n e_1 : \mathsf{snat}\ A_1 \\ \Delta;\Gamma \vdash^n e_2 : \mathsf{snat}\ A_2\end{array}}{\Delta;\Gamma \vdash^n (e_1 \hat{\circledcirc} e_2) : (\mathsf{sbool}\ (A_1 \circledcirc A_2))}\text{Arith2}$$

5. Conditional expressions.

$$e \quad ::= \mathsf{if}\ [A_1, A_2](e, X_1.e_1, X_2.e_2) \mid \cdots$$

The conditionals are again rather less conventional. In addition to the discriminated boolean expression e, it takes two type arguments A_1 and A_2, where A_1 is a proposition over booleans at type level. The second one is the proof of $A_1\ A_2$. In each of the branches of the conditional a type variable is used to which the proof $A_1\ \mathsf{true}$ (resp. $A_1\ \mathsf{false}$) is bound and thus available in the body of branch.

$$\frac{\begin{array}{ll}\Delta|_n\vdash B : \mathsf{Bool} \to \mathsf{Kind} & \Delta|_n\vdash A : (B\ A_3) \\ \Delta|_n\vdash A_2 : \Omega^{\bigcirc} & \Delta;\Gamma \vdash^n e : (\mathsf{sbool}\ A_3) \\ \Delta, X_1 : (B\ \mathsf{true})^n;\Gamma \vdash^n e_1 : A_2 & \Delta, X_2 : (B\ \mathsf{false})^n;\Gamma \vdash^n e_2 : A_2\end{array}}{\Delta;\Gamma \vdash^n (\mathsf{if}\ [B, A](e, X_1.e_1, X_2.e_2)) : A_2}\text{Cond}$$

6. Explicit staging constructs. Brackets create a piece of code, while escapes splice in a piece of code into a larger code context:

$$e \quad ::= \langle e \rangle \mid \tilde{}e \mid \cdots$$

$$\frac{\Delta;\Gamma \vdash^{n+1} e : A}{\Delta;\Gamma \vdash^n \langle e \rangle : \bigcirc A}\text{Bracket} \quad \frac{\Delta;\Gamma \vdash^n e : \bigcirc A}{\Delta;\Gamma \vdash^{n+1} \tilde{}e : A}\text{Escape}$$

7. Finally there are assert/cast constructs that are used to construct/discharge equality proofs.

$$e \quad ::= \mathsf{assert}\ e_1 : A = e_2 : B \mid \mathsf{cast}\ (e_1, B, e_2) \mid \cdots$$

$$\frac{\Delta; \Gamma \vdash^n e_1 : \mathsf{snat}\ A_1 \quad \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ A_2}{\Delta; \Gamma \vdash^n \mathsf{assert}\ e_1 : A_1 = e_2 : A_2 : (\mathsf{ID}\ A_1\ A_2)}\text{Asrt} \qquad \frac{\Delta; \Gamma \vdash^n e : (\mathsf{ID}\ A\ B) \quad \Delta; \Gamma \vdash^n e_2 : (\mathsf{snat}\ A)}{\Delta; \Gamma \vdash^n \mathsf{cast}\ (e_1, B, e_2) : (\mathsf{snat}\ B)}\text{Cast}$$

$$\begin{array}{lll} \langle\rangle|_n & = \langle\rangle & \\ \Delta, X : A^m|_n & = \begin{cases} \Delta|_n, X : A & m = n \\ \Delta|_n & \text{otherwise} \end{cases} & \\ \\ \langle\rangle^+ & = \langle\rangle & \\ (\Gamma, x : A^n)^+ & = \Gamma^+, x : A^{n+1} & \\ \\ \langle\rangle^+ & = \langle\rangle & \\ (\Delta, X : A^n)^+ & = \Delta^+, X : A^{n+1} & \end{array}$$

Figure 3.3: Typing restriction and type assignment promotion

3.3 Semantics

The semantics we shall consider here is the small step semantics (Definition 2 and Figure 3.5). The small step semantics depends on the notion of reductions (Definition 1) which relates valid redexes to their respective contractums. The small step semantics is expressed as a level-indexed family of relations between $\lambda_{H\bigcirc}$ terms (Definition 2) i.e., it describes the single-step call-by-value evaluation strategy, at a particular level, with respect to the notions of reduction.

Definition 1 (Reductions) *The notions of reduction in $\lambda_{H\bigcirc}$ are expressed by the relation $\rightarrow$ defined in Figure 3.3.*

Definition 2 (Small-step semantics) *The small step semantics of $\lambda_{H\bigcirc}$ is defined in Figure 3.5 as a relation $\_ \overset{n}{\longmapsto} \_ \subset E^n \times E^n$. The rule* $\dfrac{e_1^0 \rightarrow e_2^0}{e_1^0 \overset{0}{\longmapsto} e_2^0}$ *is intended to omit the reduction $\tilde{}\langle v^1\rangle \rightarrow v^1$, since the levels are not correct for the redex. Rather, a separate rule is added to $\overset{n}{\longmapsto}$.*

In terms of levels, the reduction relations can be divided into three groups:

$$\frac{\Delta \vdash^n \Gamma \quad (x : A^m) \in \Gamma \quad m \leq n}{\Delta; \Gamma \vdash^n x : A} \qquad \frac{\Delta \vdash^n \Gamma \text{ OK}}{\Delta; \Gamma \vdash^n m : \mathsf{snat}\, m} \qquad \frac{\Delta \vdash^n \Gamma \text{ OK}}{\Delta; \Gamma \vdash^n \mathsf{tt} : \mathsf{sbool}\ \text{True}} \qquad \frac{\Delta \vdash^n \Gamma \text{ OK}}{\Delta, \Gamma \vdash^n \mathsf{ff} : \mathsf{sbool}\ \text{False}}$$

$$\frac{\begin{array}{c}\Delta|_n\vdash A : \Omega^{\bigcirc} \\ \Delta; \Gamma, x : A^n \vdash^n f : A\end{array}}{\Delta; \Gamma \vdash^n (\mathsf{fix}\, x : A.f) \ : \ A} \qquad \frac{\Delta|_n\vdash A_1 : \Omega^{\bigcirc} \quad \Delta; \Gamma, x : A_1^n \vdash^n e : A_2}{\Delta; \Gamma \vdash^n (\lambda x : A_1.e) : A_1 \to\!\!\!\!\triangleright\ A_2} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash^n e_1 : A_1 \to\!\!\!\!\triangleright\ A_2 \\ \Delta; \Gamma \vdash^n e_2 : A_1\end{array}}{\Delta; \Gamma \vdash^n e_1\ e_2 \ : \ A_2}$$

$$\frac{\begin{array}{c}\Delta|_n\vdash B : s \\ \Delta, X : B^n; \Gamma \vdash^n f : A\end{array}}{\Delta; \Gamma \vdash^n (\Lambda X : B.f) \ : \ \forall_s X : B.A} \qquad \frac{\begin{array}{c}\Delta|_n\vdash A : B \\ \Delta, \Gamma \vdash^n e : \forall_s X : B.A_2\end{array}}{\Delta; \Gamma \vdash^n e[A] \ : \ A_2[X := A]} \qquad \frac{\Delta; \Gamma \vdash^{n+1} e : A}{\Delta; \Gamma \vdash^n \langle e \rangle : \bigcirc A} \qquad \frac{\Delta; \Gamma \vdash^n e : \bigcirc A}{\Delta; \Gamma \vdash^{n+1} \tilde{} e : A}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^n e_1 : \mathsf{snat}\ A_1 \\ \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ A_2\end{array}}{\Delta; \Gamma \vdash^n e_1 \oplus e_2 : \mathsf{snat}\ (A_1 \hat{\oplus} A_2)} \qquad \frac{\begin{array}{c}\Delta|_n\vdash A : B \\ \Delta|_n\vdash B : s \\ \Delta; \Gamma \vdash^n e : A[X := A_1]\end{array}}{\Delta; \Gamma \vdash^n \{X = A_1, e : A\} : \exists_s X : B.A}$$

$$\frac{\begin{array}{l}\Delta, \Gamma \vdash^n e : \exists_s X' : B.A_1 \\ \Delta|_n\vdash A_2 : \Omega^{\bigcirc} \\ \Delta, X : B; \Gamma, x : A_1[X' := X] \vdash^n e_2 : A_2\end{array}}{\Delta; \Gamma \vdash^n \mathsf{open}\ e_1\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2 : A_2} X \notin \Delta \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash^n e_1 : \mathsf{tup}\ A_3\ B \\ \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ A_2 \\ \Delta|_n\vdash A : \mathsf{LT}\ A_2\ A_3\end{array}}{\Delta; \Gamma \vdash^n \mathsf{sel}\,[A](e_1, e_2) : B\ A_2}$$

$$\frac{\begin{array}{ll}\Delta|_n\vdash B : \text{Bool} \to Kind & \Delta; \Gamma \vdash^n e : \mathsf{sbool}\ A_3 \\ \Delta|_n\vdash A : B\ A_3 & \Delta, X_1 : B\ \mathsf{true}; \Gamma \vdash e_1 : A_2 \\ \Delta|_n\vdash A_2 : \Omega^{\bigcirc} & \Delta, X_2 : B\ \mathsf{false}; \Gamma \vdash e_2 : A_2\end{array}}{\Delta; \Gamma \vdash^n \mathsf{if}\ [B, A](e, X_1.e_1, X_2.e_2) : A_2} \qquad \frac{0 \leq i < m.\ \Delta, \Gamma \vdash^n e_i : A_i}{\Delta; \Gamma \vdash^n (e_0, \ldots, e_{m-1}) : \mathsf{tup}\ \hat{m}\ (\mathsf{nth}\ [A_0, \ldots, A_{m-1}])}$$

$$\frac{\Delta; \Gamma \vdash^n e : A_1 \quad A_1 = A_2 \quad \Delta|_n\vdash A_2 : \Omega^{\bigcirc}}{\Delta; \Gamma \vdash^n e : A_2} \qquad \frac{\Delta; \Gamma \vdash^n e_1 : \mathsf{snat}\ A \quad \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ B}{\Delta; \Gamma \vdash^n \mathsf{assert}\ e_1 : A = e_2 : B : \text{ID}\ A\ B}$$

$$\frac{\Delta; \Gamma \vdash^n e_1 : \text{ID}\ A\ B \quad \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ A}{\Delta; \Gamma \vdash^n \mathsf{cast}\ (e_1, B, e_2) : \mathsf{snat}\ B}$$

Figure 3.4: Type system of $\lambda_{H\bigcirc}$

1. Most active reduction steps, such as beta and delta reductions, occur at level 0 ($\overset{0}{\longmapsto}$). The rule for bracketed expressions forces the reduction of the expression inside the brackets to be reduced at a higher level: $\dfrac{e_1 \overset{n+1}{\longmapsto} e_2}{\langle e_1 \rangle \overset{n}{\longmapsto} \langle e_2 \rangle}$.

2. At level 1 ($\overset{1}{\longmapsto}$), escapes are performed: $\dfrac{}{\tilde{} v^1 \overset{1}{\longmapsto} v^1}$.

3. At level $n \geq 2$ expressions are simply rebuilt. Escaped expressions are reduced at a lower level, $\dfrac{e_1 \overset{n+1}{\longmapsto} e_2}{\tilde{} e^{n+1} \overset{n+2}{\longmapsto} \tilde{} e_2}$.

Definition 3 (Termination) *Let $e \in E^n$ be an expression.*

1. *Termination: $e \Downarrow^n$ iff $\exists v \in V^n.\, e \overset{n}{\longmapsto} v$*

2. *Non-termination: $e \Uparrow^n$ iff $\forall e'.e \overset{n}{\longmapsto} e' \Rightarrow \exists e''.e' \overset{n}{\longmapsto} e''$.*

$$\frac{e_1^0 \to e_2^0}{e_1^0 \overset{0}{\longmapsto} e_2^0} \qquad \frac{}{\tilde{}\langle v^1\rangle \overset{1}{\longmapsto} v^1} \qquad \frac{e_1^{n+1} \overset{n+1}{\longmapsto} e_2^{n+1}}{\lambda x.e_1^{n+1} \overset{n+}{\longmapsto} \lambda x.e_2^{n+1}} \qquad \frac{e_1^n \overset{n}{\longmapsto} e_3^n}{e_1^n\ e_2^n \overset{n}{\longmapsto} e_3^n\ e_2^n} \qquad \frac{e_1^n \overset{n}{\longmapsto} e_2^n}{v^n\ e_1 \overset{n}{\longmapsto} v^n\ e_2^n} \qquad \frac{e_1^n \overset{n}{\longmapsto} e_2^n}{e_1^n[A] \overset{n}{\longmapsto} e_2^n[A]}$$

$$\frac{f^{n+1} \overset{n+1}{\longmapsto} f_2^{n+}}{\mathsf{fix}\, x : A.f_1^{n+1} \overset{n+1}{\longmapsto} \mathsf{fix}\, x : A.f_2^{n+1}} \qquad \frac{e_1^n \overset{n}{\longmapsto} e_2^n}{\{X = A_1, e_1^n : A_2\} \overset{n}{\longmapsto} \{X = A_1, e_2^n : A_2\}} \qquad \frac{e_1^{n+1} \overset{n+1}{\longmapsto} e_2^{n+1}}{\langle e^{n+1}\rangle \overset{n}{\longmapsto} \langle e_2^{n+1}\rangle} \qquad \frac{e_1^{n+1} \overset{n+1}{\longmapsto} e_2^{n+1}}{\tilde{}e_1^{n+1} \overset{n+2}{\longmapsto} \tilde{}e_2^{n+1}}$$

$$\frac{e_1^n \overset{n}{\longmapsto} e_3^n}{\mathsf{open}\ e_1^n\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2^n \overset{n}{\longmapsto} \mathsf{open}\ e_3^n\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2^n} \qquad \frac{e_2^{n+1} \overset{n+1}{\longmapsto} e_3^{n+1}}{\mathsf{open}\ v_1^{n+1}\ \mathsf{as}\ X, x\ \mathsf{in}\ e_2^{n+1} \overset{n+1}{\longmapsto} \mathsf{open}\ v_3^{n+1}\ \mathsf{as}\ X, x\ \mathsf{in}\ e_3^{n+1}}$$

$$\frac{e^n \overset{n}{\longmapsto} e_3^n}{\mathsf{if}\ [B, A](e^n, X_1.e_1^n, X_2.e_1^n) \overset{n}{\longmapsto} \mathsf{if}\ [B, A](e_3^n, X_1.e_1^n, X_2.e_2^n)}$$

$$\frac{e_1^{n+1} \overset{n+1}{\longmapsto} e_3^{n+1}}{\mathsf{if}\ [B, A](v^{n+1}, X_1.e_1^{n+1}, X_2.e_2^{n+1}) \overset{n+1}{\longmapsto} \mathsf{if}\ [B, A](v^{n+1}, X_1.e_3^{n+1}, X_2.e_2^{n+1})}$$

$$\frac{e_2^{n+1} \overset{n+1}{\longmapsto} e_3^{n+1}}{\mathsf{if}\ [B, A](v^{n+1}, X_1.v_1^{n+1}, X_2.e_2^{n+1}) \overset{n+1}{\longmapsto} \mathsf{if}\ [B, A](v^{n+1}, X_1.v_1^{n+1}, X_2.e_3^{n+1})}$$

$$\frac{e_1^n \overset{n}{\longmapsto} e_2^n}{\mathsf{sel}\,[A](e_1^n, e_2^n) \overset{n}{\longmapsto} \mathsf{sel}\,[A](e_3^n, e_2^n)} \qquad \frac{e_1^n \overset{n}{\longmapsto} e_2^n}{\mathsf{sel}\,[A](v^n, e_1^n) \overset{n}{\longmapsto} \mathsf{sel}\,[A](v^n, e_2^n)}$$

$$\frac{f_1^{n+1} \overset{n+1}{\longmapsto} f_2^{n+1}}{\Lambda X : B.f_1^{n+1} \overset{n+1}{\longmapsto} \Lambda X : B.f_2^{n+1}} \qquad \frac{e^n \overset{n}{\longmapsto} e'^n}{(v_0^n, \ldots, v_{i-1}^n, e^n, \ldots, e_m^n) \overset{n}{\longmapsto} (v_0^n, \ldots, v_{i-1}^n, e'^n, \ldots, e_m^n)}$$

$$\frac{e_1^n \overset{n}{\longmapsto} e_1'^n}{\mathsf{assert}\ e_1^n : A = e_2^n : B \overset{n}{\longmapsto} \mathsf{assert}\ e_1'^n : A = e_2^n : B} \qquad \frac{e_2^n \overset{n}{\longmapsto} e_2'^n}{\mathsf{assert}\ v_1^n : A = e_2^n : B \overset{n}{\longmapsto} \mathsf{assert}\ v_1^n : A = e_2'^n : B}$$

$$\frac{e_1^n \overset{n}{\longmapsto} e_1'^n}{\mathsf{cast}\ (e_1^n, B, e_2) \overset{n}{\longmapsto} \mathsf{cast}\ (e_1'^n, B, e_2)}$$

Figure 3.5: Small step semantics of $\lambda_{H\bigcirc}$

A Note on `assert/cast` The construct $\mathsf{assert}\ e_1 : A = e_2 : B$ introduces a term of equality type $\mathsf{ID}\ A\ B$ provided that e_1 and e_2 are A and B snats, respectively. The semantics of `assert` is perhaps the most difficult one to understand: there is only one form of `assert` value, $\mathsf{assert}\ v : A = v : B$, i.e., only that where its argument values are equal. Otherwise, if the two values are not equal, the assert expression reduces to the non-terminating expression $\Omega_{\mathsf{ID}\ A\ B}$. [3] This is done in order to preserve the progress property, i.e., even if the asserted values are not equal, the system will not get stuck: rather failure of assertion is modeled by non-termination as embodied by the Ω term. (This should not be confused with the name of $\lambda_{H\bigcirc}$ types $\Omega^{\bigcirc}$.)

Similarly, the semantics of $\mathsf{cast}\ (e_1, B, e_2)$ must first evaluate its first argument e_1 (which is presumed to be an `assert`). Only if a value is obtained (i.e., assertion has not failed), the reduction rule simply eliminates the cast and proceeds to e_2.

[3] Incidentally, this is why the types must be carried with assert, in order to instantiate the Ω expression to the appropriate type.

$$
\begin{array}{ll}
(\lambda x : A.e^0)\ v^0 & \longrightarrow e^0[x := v^0] \\
(\Lambda X : B.f^0)[A] & \longrightarrow f^0[X := A] \\
\mathsf{sel}\,[A_1]((v_0^0, \ldots, v_n^0), m) & \longrightarrow v_m\ \mathsf{if} m < n \\
\mathsf{open}\ \{X_1 = A_1, v^0 : A_2\}\ \mathsf{as}\ X, x\ \mathsf{in}\ e^0 & \longrightarrow e^0[X := A_1][x := v^0] \\
(\mathsf{fix}\, x : A.f^0)\ v^0 & \longrightarrow f^0[x := \mathsf{fix}\, x : A.f^0]\ v^0 \\
(\mathsf{fix}\, x : A.f^0)[A_2] & \longrightarrow f^0[x := \mathsf{fix}\, x : A.f^0]\ [A_2] \\
m \oplus n & \longrightarrow m \overline{\oplus} n \\
\mathsf{if}\ [B, A](\mathsf{tt}, X_1.e_1, X_2.e_2) & \longrightarrow e_1[X_1 := A] \\
\mathsf{if}\ [B, A](\mathsf{ff}, X_1.e_1, X_2.e_2) & \longrightarrow e_2[X_2 := A] \\
\tilde{}\langle v^1 \rangle & \longrightarrow v^1 \\
\mathsf{assert}\ v_1^0 : A = v_2^0 : B & \longrightarrow \Omega_{\mathsf{ID}\ A\ B}\ \mathsf{if} v_1^0 \neq v_2^0 \\
\mathsf{cast}\ (v^0, B, e^0) & \longrightarrow e^0
\end{array}
$$

$$\Omega_A \equiv (\mathsf{fix}\ f : () \to A.\lambda y : ().f\ y)\ ()$$

Figure 3.6: Reductions of $\lambda_{H\bigcirc}$

3.4 Properties of $\lambda_{H\bigcirc}$

In this section we will sketch out and develop the proof of the main technical result we report here, the type safety of $\Omega^{\bigcirc}$ (Theorem 1). For this proof, we adapt a standard syntactic techniques of Wright and Felleisen [145].

Theorem 1 (Type safety) *If* $\Delta; \Gamma^+ \vdash^n e^n : A$ *then* $e \overset{n}{\longmapsto}^* v^n$, *and* $\Delta, \Gamma^+ \vdash^n v : A$, *or* $e \Uparrow$.

PROOF. First, we establish the subject reduction property of the reductions of $\lambda_{H\bigcirc}$ (Lemma 5). This can easily be generalized to the subject-reduction of the small-step reduction relation. Secondly, we establish the progress property of the small-step reduction relation (Lemma 1). Type safety property follows quite easily from these [145]. □

In proofs of critical lemmas, we shall need a property of TL (Remark 2) typing judgments observed by Shao&al. [115].[4]

Remark 2 (Judgment normal forms [116]) *Due to transitivity of conversion, any derivation of* $\Delta; \Gamma \vdash^n e : A$ *can be converted into a normal form such that*

1. *The root node of the derivation is a CONV rule.*
2. *Its first derivation ends with a rule other than CONV.*
3. *All of whose term derivations are in the normal form.*

[4] We omit the proof (by transitivity of $=_{\beta\eta\iota}$ of TL and induction on the structure of typing judgments).

Lemma 1 (Progress) *If* $\Delta; \Gamma^+ \vdash^n e^n : A$, *then* $e^n \in V^n$ *or* $\exists e'.e \longmapsto e'$.

PROOF. Proof is by structural induction on the judgment $e \in E^n$, and then by examination of cases on the typing judgment $\Delta; \Gamma^+ \vdash^n e^n : A$. We show some of the relevant cases.

1. *Variable case,* $e^n = x$. There are two cases on n:
   (a) Case $n = 0$. If $n = 0$, then $\neg(\Delta; \Gamma^+ \vdash^0 x : A)$, since the levels of all variables in Γ^+ are greater than 0.
   (b) Case $n > 0$. Then, by definition of $\in V^n$, $x \in V^n$ for all $n > 0$.
2. *Constant case,* $e \in m, \mathsf{tt}, \mathsf{ff}$. This follows trivially, by definition of $\in V^n$, since all the constants are already in V^n, for all n.
3. *Function abstraction case,* $e = \lambda x : A.e$. Let us consider the normal form for the derivation $\Delta; \Gamma^+ \vdash^n (\lambda x : A.e) : (A \to A')$. The derivation of $\Delta; \Gamma^+ \vdash^n \lambda x : A.e^n : A' \to A_2$ has a subderivation $\Delta; \Gamma^+, x : A_1^{n+1} \vdash^n e^n : A_3$,where $A_3 = A_2$. By the inductive hypothesis there are again two possibilities:
   - $e^n \in V^n$. Then, easily $\lambda x : A_1.e^n \in V^n$.
   - $\exists e'.e^n \overset{n}{\longmapsto} e'$. Then, by definition of $\overset{n}{\longmapsto}$, $\exists e'' = \lambda x : A_1.e'$ and $e \overset{n}{\longmapsto} e''$.
4. *Fix case,* $e = \mathsf{fix}\, x : A.f^n$.
   In the premise of the root of the derivation $\Delta; \Gamma^+ \vdash^n (\mathsf{fix}\, x : A.f^n) : A$ must have been derived by the Fix rule, with the hypothesis $\Delta; \Gamma^+, x : A^{n+1} \vdash^n f^n : A_2$ (where $A_2 = A$). We can apply the inductive hypothesis to this sub-derivation and have two possibilities:
   (a) $f^n \in V^n$, from which it immediately follows by definition of V^n that $\mathsf{fix}\, x : A.f^n \in V^n$
   (b) $\exists e'.f^n.f^n \overset{n}{\longmapsto} e'$, from which it follows by definition of $\overset{n}{\longmapsto}$ that $\mathsf{fix}\, x : A.f^n \overset{n}{\longmapsto} \mathsf{fix}\, x : A.f^n$.
5. *Code-bracket case,* $e = \langle e^{n+1} \rangle$. Then an antecedent of $\Delta; \Gamma^+ \vdash^n \langle e^{n+1} \rangle : \bigcirc A$ must have been $\Delta; \Gamma^+ \vdash^{n+1} e^{n+1} : A_1$ (where $A_1 = A$). We apply the inductive hypothesis, and examine two cases
   (a) $e^n \in V^n$, then $\langle e^n \rangle \in V^n$ by definition of V^n.
   (b) $\exists e'.e^n \overset{n+1}{\longmapsto} e'$ Then, by definition of $\dot{\longmapsto}$, $\langle e^{n+1} \rangle \overset{n}{\longmapsto} \langle e'' \rangle$.
6. *Escape at level 1,* $e^1 = \tilde{} e^0$. The type derivation looks as follows(in normal form):

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Delta; \Gamma^+ \vdash^0 e^0 : \bigcirc A_1}\text{Esc}}{\Delta; \Gamma^+ \vdash^1 \tilde{} e^0 : A_1} \quad A_1 = A}{\Delta; \Gamma^+ \vdash^1 \tilde{} e^0 : A}\text{CNV}$$

The induction hypothesis applies to the result of the conclusion of the subderivation $\mathcal{D}$. There are two possibilities:

(a) $e^0 \in V^0$ It is easily shown (by examination of cases and the type judgment rules) that the only value at level 0 that could have type $\bigcirc A_1$ is of the form $\langle v^1 \rangle$. Then, by definition of $\in V^1$, $\tilde{}\langle v^1 \rangle \in V^1$.

(b) $\exists e'.e^0 \stackrel{0}{\longmapsto} e'$. Then, by definition of $\stackrel{1}{\longmapsto}$, $\tilde{}e^0 \stackrel{1}{\longmapsto} \tilde{}e'$.

7. *Escape at higher level case,* $e^{n+2} = \tilde{}e^{n+1}$.

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Delta;\Gamma^+ \vdash^{n+1} e : \bigcirc A_1}\text{CNV}}{\Delta;\Gamma^+ \vdash^{n+2} \tilde{}e : A_1 \;\; A_1 = A}\text{Esc}}{\Delta;\Gamma^+ \vdash^{n+2} \tilde{}e : A}\text{CNV}$$

We can apply the induction hypothesis to $\Delta;\Gamma^+ \vdash^{n+1} e : \bigcirc A_1$. There are two possibilities now,

(a) $e^{n+1} \in V^{n+1}$. Then, by definition of V^{n+2}, $\tilde{}e^{n+1} \in V^{n+2}$

(b) $\exists e'.e^{n+1} \stackrel{n+1}{\longmapsto} e'$. Then, by definition of $\longmapsto$, $\tilde{}e^{n+1} \stackrel{n+2}{\longmapsto} \tilde{}e'$.

8. *The assert case,* $e^n = \mathsf{assert}\ e_1^n : A = e_2^n : B$. The typing derivation can be put into following normal form

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_1}{\Delta;\Gamma^+ \vdash^n e_1^n : A'}\text{CNV} \quad \dfrac{\mathcal{D}_2}{\Delta;\Gamma^+ \vdash^n e_2^n : B'}\text{CNV}}{\Delta;\Gamma^+ \vdash^n (\mathsf{assert}\ e_1^n : A = e_2^n : B) : (\mathsf{ID}\ A'\ B') \quad A = A' \wedge B = B'}\text{ASSRT}}{\Delta;\Gamma^+ \vdash^n (\mathsf{assert}\ e_1^n : A = e_2^n : B) : (\mathsf{ID}\ A\ B)}\text{CNV}$$

Now, we can apply the induction hypothesis to subjudgments $\dfrac{\mathcal{D}_1}{\Delta;\Gamma^+ \vdash^n e_1^n : A'}\text{CNV}$ and $\dfrac{\mathcal{D}_2}{\Delta;\Gamma^+ \vdash^n e_2^n : B'}\text{CNV}$ to obtain $e_1^n \in V^n \vee \exists e_1'.e_1^n \stackrel{n}{\longmapsto} e_1'$ and $e_2^n \in V^n \vee \exists e_2'.\longmapsto e_2^n \stackrel{n}{\longmapsto} e_2'$. We examine the cases that arise one by one.

(a) Case $e_1 \in V^n$ and $e_2^n \in V^n$. If $n > 0$, then trivially $\mathsf{assert}\ e_1 : A = e_2 : B \in V^n$. Otherwise, if $n = 0$, there are two possibilities. First $e_1 = e_2$ in which case $\mathsf{assert}\ e_1 : A = e_2 : B \in V^0$ by definition. If they are not equal, however, there exists $e'' = \Omega_{\mathsf{ID}\ A\ B}$ to which $\mathsf{assert}\ e_1 : A = e_2 : B$ reduces.

(b) Case $e_1 \in V^n$ and $\exists e_2'.e_2 \stackrel{n}{\longmapsto} e_2'$. Then by definition of $\stackrel{n}{\longmapsto}$, $\exists e'' = \mathsf{assert}\ e_1 : A = e_2' : B$ such that $\mathsf{assert}\ e_1 : A = e_2 : B \stackrel{n}{\longmapsto} e''$.

(c) Case $\exists e_1'.e_1 \stackrel{n}{\longmapsto} e_1'$ and $e_2 \in V^n$. Then, as in previous case $\exists e'' = \mathsf{assert}\ e_1' : A = e_2 : B$ such that $\mathsf{assert}\ e_1 : A = e_2 : B \stackrel{n}{\longmapsto} e''$.

(d) Case $\exists e_1'.e_1^n \overset{n}{\longmapsto} e_1'$ and $\exists e_2'. \longmapsto e_2^n \overset{n}{\longmapsto} e_2'$. Then let e'' be $\mathsf{assert}\ e_1' : A = e_2 : B$ and by definition of $\overset{n}{\longmapsto}$, $\mathsf{assert}\ e_1 : A = e_2 : B \overset{n}{\longmapsto} e''$.

9. *Cast case,* $e^n = \mathsf{cast}\ (e_1^n, B, e_2^n)$. This case is more interesting. The typing judgment for e^n can be put into the following normal form:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}_1}{\Delta;\Gamma^+ \vdash^n e_1 : \mathsf{ID}\ A\ B'}\text{CNV} \quad \dfrac{\mathcal{D}_2}{\Delta;\Gamma^+ \vdash^n e_2^n : \mathsf{snat}\ A'}\text{CNV}}{\Delta;\Gamma^+ \vdash^n (\mathsf{cast}\ (e_1^n, B, e_2^n)) : (\mathsf{snat}\ B') \quad B = B'}\text{CAST}}{\Delta;\Gamma^+ \vdash^n (\mathsf{cast}\ (e_1^n, B, e_2^n)) : (\mathsf{snat}\ B)}\text{CNV}$$

We can apply the induction hypothesis to subderivations $\dfrac{\mathcal{D}_1}{\Delta;\Gamma^+ \vdash^n e_1 : \mathsf{ID}\ A'\ B'}$ and $\dfrac{\mathcal{D}_2}{\Delta;\Gamma^+ \vdash^n e_2^n : \mathsf{snat}\ A'}$. We examine the cases that arise.

(a) Case $e_1 \in V^n$ and $e_2 \in V^n$. First, if $n > 0$, then trivially $\mathsf{cast}\ (e_1, B, e_2) \in V^n$. Otherwise, if $n = 0$, then $\exists e' = e_2$ such that by definition of reduction $\mathsf{cast}\ (e_1, B, e_2) \overset{0}{\longmapsto} e'$.

(b) Case $e_1 \in V^n$ and $\exists e'.e_2 \overset{n}{\longmapsto} e'$. Then, same as above by definition of the reduction relation.

(c) Case $\exists e_1'.e_1 \overset{n}{\longmapsto} e_1'$ and $e_2 \in V^n$. Then $\exists e'' = \mathsf{cast}\ (e_1', B, e_2)$ such that $\mathsf{cast}\ (e_1, B, e_2) \overset{n}{\longmapsto} e''$ by definition of $\overset{n}{\longmapsto}$.

(d) Case $\exists e_1'.e_1 \overset{n}{\longmapsto} e_1'$ and $\exists e_2'.e_2 \overset{n}{\longmapsto} e_2'$. Similar as above case.

□

Lemma 2 (Level Increment) *If* $\Delta;\Gamma \vdash^n e : A$*, then* $\Delta;\Gamma^+ \vdash^{n+1} e : A$.

PROOF. Proof of Lemma 3.4 is by induction on height of type derivations of $\Delta;\Gamma \vdash^n e : A$. □

Lemma 3 (Substitution 1) *If* $\Delta;\Gamma, x : A^m, \Gamma' \vdash^n e : B$ *and* $\Delta;\Gamma,\Gamma' \vdash^m e_2 : A$*, then* $\Delta;\Gamma,\Gamma' \vdash^n e[x := e_2] : B$.

Lemma 4 (Substitution 2) *If* $\Delta, X : B^n;\Gamma \vdash^n e : A_1$ *and* $\Delta|_n \vdash^n A_2 : B$*, then* $\Delta;\Gamma[X := A_2] \vdash^n e : A_1[X := A_2]$.

PROOF. Proof of Lemma 3 is by induction on the type derivation. Also, Lemma 3.4 is used to prove the base case.

Similarly, Lemma 4 is by induction on type derivations. □

Lemma 5 (Subject Reduction) $\forall n.$ *if* $\Delta, \Gamma^+ \vdash^n e : A$ *and* $e \to e'$*, then* $\Delta, \Gamma^+ \vdash^n e' : A$*.*

PROOF. Proof is by examination of cases of possible reductions $e \to e'$.

1. Beta reduction, $(\lambda x : A.e^0)\ v^0 \to e^0[x := v^0]$.

   Consider the normal form of the typing judgment for the redex:

   $$\dfrac{\dfrac{\dfrac{D_1}{\Delta, \Gamma \vdash^0 (\lambda x : A.e^0) : A \to A_1} \quad \dfrac{D_2}{\Delta, \Gamma \vdash^0 v^0 : A}\text{App} \quad A_1 = A_2}{\Delta, \Gamma \vdash^0 (\lambda x : A.e^0)\ v^0 : A_1}\text{CNV}}{\Delta, \Gamma \vdash^0 (\lambda x : A.e^0)\ v^0 : A_2}$$

   Applying substitution lemma 1, we have $\dfrac{\Delta, \Gamma \vdash^0 e^0[x := v^0] : A_1 \quad A_1 = A_2}{\Delta, \Gamma \vdash^0 e^0[x := v^0] : A_2}\text{CNV}$.

2. $(\Lambda X : A.f^0)[A_2] \to f^0[X := A_2]$ The derivation for the redex can be put into following normal form: $\dfrac{\dfrac{\dfrac{\dfrac{D_1}{\Delta, X : A; \Gamma \vdash^0 f^0 : B_2}}{\Delta, \Gamma \vdash^o (\Lambda_s X : A.f^0) : \forall_s X : A.B_2} \quad \dfrac{D_2}{\Delta, \Gamma \vdash^n A_2 : A}\text{TyApp}}{\Delta; \Gamma \vdash^0 (\Lambda X : A.f^0)[A_2] : B_2[X := A_2]}\text{CONV}}{\Delta; \Gamma \vdash^0 (\Lambda X : A.f^0)[A_2] : B[X := A_2]}$

   Applying the substitution lemma (Lemma 4) we obtain $\Delta, \Gamma[X := A] \vdash^n f^0[X := A] : B_2[X := A]$. But then, $\Delta, \Gamma[X := A] \vdash^n f^0[X := A] : B[X := A]$, since $B = B_2$.

   Since $\Delta, \Gamma \vdash^0 (\Lambda_s X : A.f^0) : \forall_s X : A.B_2$, then and Γ is well formed, then we must conclude that $X \notin FV(\Gamma)$, and so the substitution $\Gamma[X := A] = \Gamma$. Then, from this we easily conclude that $\Delta; \Gamma \vdash f[X := A] : B[X := A]$.

3. $\mathsf{sel}\,[A]((v_0^0, \ldots, v_n^0), m) \to v_m$, if $m < n$

   The proof of this case is essentially unchanged from the proof in [116]. The type derivation of the redex can be put into the following normal form:

   $$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{\forall i < m. \vdash v_i : A_i''\ i}}{\Delta; \Gamma^+ \vdash \overline{v} : \mathsf{tup}\ n\ A_1''} \quad \dfrac{\mathcal{E}}{\Delta; \Gamma^+ \vdash m^n : \mathsf{snat}\ m} \quad \dfrac{\mathcal{F}}{\Delta; \Gamma^+ \vdash^n A' : \mathsf{LT}\ A_1\ A_2}}{\Delta; \Gamma^+ \vdash^n \mathsf{sel}[A']((v_0, \ldots, v_n), m) : A'' A_1 \quad A'' A_1 = A}}{\Delta; \Gamma^+ \vdash^n \mathsf{sel}[A']((v_0, \ldots v_n), m) : A}$$

   Here, $A =_{\beta\eta\iota} A''\ A_1$,$A_1'' = A''$, and $A_1 = m$. By examining the reduction, we have $m < n$. The redex reduces to the value v_m. $A_1''\ m = A$, and $\Delta; \Gamma^+ \vdash^n D_m : A_1''\ m$, we obtain $\Delta; \Gamma^+ \vdash^n v_m : A$.

4. $\mathsf{open}\ \{Y = A_1, v^0 : A_2\}\ \mathsf{as}\ X, x\ \mathsf{in}\ e^0 \to e^0[X := A_1][x := v^0]$ (check)

   The derivation of the redex term (taking into consideration conversion and normal form) [e.p. too large to fit in here]: IF, $\Delta; \Gamma \vdash^0 \mathsf{open}\ \{Y = A_1, v^0 : A_2\}\ \mathsf{as}\ X, x\ \mathsf{in}\ e^0 : C$, then

- 4: $\Delta; \Gamma \vdash^0 v^0 : A_2[Y := A_1]$
- 4: $\Delta, X : B^0; \Gamma, x : A_2[Y := X] \vdash^0 e^0 : C$
- 4: $\Delta|_0 \vdash C : \Omega^{\bigcirc}$

Applying the substitution lemma for types to 4, we obtain:

$$\Delta; (\Gamma, x : A_2[Y := X])[X := A_1] \vdash^0 e^0[X := A_1] : C[X := A_1]$$

Since $\Delta|_0 \vdash C : \Omega^{\bigcirc}$, it can be easily shown that $X \notin FV(C)$. Thus the above expression can be simplified to (by definition of substitution):

$$\begin{array}{rl} & \Delta; (\Gamma, x : A_2[Y := X])[X := A_1] \vdash^0 e^0[X := A_1] : C \\ = & \Delta; \Gamma[X := A_1], x : A_2[X := A_1] \vdash^0 e^0[X := A_1] : C \\ = & \Delta; \Gamma, x : A_2[X := A_1] \vdash^0 e[X := A_1] : C \end{array}$$

Now, we apply the substitution lemma for terms (using the fact of 4) to obtain the typing from the contractum:

$$\Delta; \Gamma \vdash^0 e[X := A_1][x := v^0] : C$$

5. $(\mathsf{fix}\, x : A.f^0)\ v^0 \to f^0[x := \mathsf{fix}\, x : A.f^0]\ v^0$ By substitution lemma for terms.
6. $(\mathsf{fix}\, x : A.f^0)[A_2] \to f^0[x := \mathsf{fix}\, x : A.f^0]\ [A_2]$ By substitution lemma for types.
7. $m \oplus n \to m \overline{\oplus} n$

$$\frac{\mathcal{D}}{\Delta; \Gamma \vdash^0 i \oplus j : \mathsf{snat}\ i \hat{\oplus} j}$$

   By the adequacy of TL representation of arithmetic ([116],Lemma 1) it easily follows $\Delta; \Gamma \vdash^0 i \overline{\oplus} j : \mathsf{snat}\ i \hat{\oplus} j$.

8. $\mathsf{if}\ [B, A](\mathsf{tt}, X_1.e_1^0, X_2.e_1^0) \to e_1^0[X_1 := A]$ (proven in the paper[115]. same proof)
9. $\mathsf{if}\ [B, A](\mathsf{ff}, X_1.e_1^0, X_2.e_1^0) \to e_2^0[X_2 := A]$ (proven in the paper[115]. same proof)
10. $\tilde{}\langle v^1 \rangle \to v^1$ The type derivation for the redex $\tilde{}\langle v^1 \rangle$ can be put into the following normal form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{D}{\Delta; \Gamma \vdash^1 v^1 : A_3}}{\Delta; \Gamma \vdash^0 \langle v^1 \rangle : \bigcirc A_3}\mathrm{Br} \quad A_2 = A_3}{\Delta; \Gamma \vdash^0 \langle v^1 \rangle : \bigcirc A_2}\mathrm{CNV}}{\Delta; \Gamma \vdash^1 es\langle v^1 \rangle : A_2}\mathrm{Esc} \quad A_2 = A}{\Delta; \Gamma \vdash^1 \tilde{}\langle v^1 \rangle : A}\mathrm{CNV}$$

Immediately, from the conclusion of subderivation D it follows that

$$\dfrac{\dfrac{\dfrac{D}{\Delta;\Gamma \vdash^n v^1 : A_3}\text{CNV} \quad A_3 = A_2}{\Delta;\Gamma \vdash^n v^1 : A_2}\text{CNV} \quad A_2 = A}{\Delta;\Gamma \vdash^1 v^1 : A}\text{CNV}$$

11. Reduction $\mathsf{assert}\ v_1^0 : A = v_2^0 : B \to \Omega_{\mathsf{ID}\ A\ B}$ if $v_1 \neq v_2$. Immediately, we can construct a derivation of $\mathsf{ID}\ A\ B$ for $\Omega_{\mathsf{ID}\ A\ B}$ as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta;\Gamma \vdash (\mathsf{fix}\, f : () \to \mathsf{ID}\ A\ B.\lambda y : ().f\ y) : () \to \mathsf{ID}\ A\ B}\text{FIX} \qquad \dfrac{}{\Delta;\Gamma \vdash () : ()}}{\Delta;\Gamma \vdash (\mathsf{fix}\, f : () \to \mathsf{ID}\ A\ B.\lambda y : ().f\ y)\ ()}\text{APP}$$

12. The case of the reduction $\mathsf{cast}\ (v^0, B, e^0) \to e^0$ is the most interesting one. The normal form of the derivation for the subject term is as follows:

$$\dfrac{\dfrac{\mathcal{D}_1}{\Delta;\Gamma \vdash^0 e_1 : \mathsf{ID}\ A\ B} \qquad \dfrac{\mathcal{D}_2}{\Delta;\Gamma \vdash^0 e_2 : \mathsf{snat}\ A}}{\Delta;\Gamma \vdash^0 \mathsf{cast}\ (v, B, e) : (\mathsf{snat}\ B)}\text{CNV}$$

We must show that $\Delta;\Gamma \vdash^0 e : \mathsf{snat}B$. At first, this would seem very difficult because we have no proof that A reduces to B. However, since v is a value of type $\mathsf{ID}\ A\ B$, its first argument must have been of type $\mathsf{snat}B$.

Now, by adequacy on equality of values, we know that $\forall v_1 : \mathsf{snat}\ A.\forall v_2 : \mathsf{snat}\ B.v_1 = v_2 \Rightarrow A =_{\beta\eta\iota} B$. Then it is possible to use CNV to construct the derivation $\Delta;\Gamma \vdash^0 e : \mathsf{snat}\ B$.

□

Lemma 6 (Subject reduction $(\dot{\longmapsto})$) *$\forall n \in \mathbb{N}.\ \Delta, \Gamma^+ \vdash^n e : A$ and $e \overset{n}{\longmapsto} e'$, then $\Delta;\Gamma^+ \vdash^n e' : A$.*

PROOF. Proof follows easily from Lemma 5 and induction on the height of derivations of $\overset{n}{\longmapsto}$. □

3.5 Conclusion

In this section we have presented a calculus that combines type theoretic features such as singleton types and equality assertions with staging. While this language is not identical with Meta-D, we conjecture that its extension to full Meta-D features is possible (though quite tedious in practice). However, having proved type safety of such a language we have, in principle, showed that it is plausible to combine a form of dependent typing and staging in a programming language.

Furthermore, we believe that the route we took in our examination of $\lambda_{H\bigcirc}$ could have important practical benefits. Namely, since $\lambda_{H\bigcirc}$ is defined as one of the computational languages in the FLINT framework, it should, in principle, be possible to use any future FLINT implementation to provide a general infrastructure for the implementation of programming languages with features similar to $\lambda_{H\bigcirc}$. We have not experimented with adding $\lambda_{H\bigcirc}$ to the FLINT compiler, but we consider it an interesting direction for future work.

Part II

Open Heterogeneous Meta-programming in Haskell

Chapter 4

Equality Proofs

4.1 Introduction

In Chapter 2 we have explored the practice of heterogeneous meta-programming with dependent types. In this chapter we shall develop the ideas presented earlier in a different setting: we will show that a heterogeneous meta-programming framework can be implemented in a functional language with higher-rank polymorphism. The meta-language under consideration is strikingly similar to current dialects of Haskell, but with a few key extensions.

Outline of this chapter. This chapter is organized as follows. First, we describe a technique for implementing equality proofs between types in Haskell-like languages (Section 4.2). Then we illustrate how equality proofs can be used to encode *domain values*, and predicates in Haskell's type system. We develop an example that defines arithmetic operators on natural number domain values, and encodes a couple of interesting predicates over over natural numbers (Section 4.3).

4.2 Type Equality

One can view a language such as Haskell from the perspective of the Curry-Howard [60, 45] isomorphism: types are proposition in a logical language (where types are formulas); programs that inhabit particular types are proofs of the propositions that correspond to their types. For example, the function $\lambda x : \mathsf{Int}.\, x$ is the proof of the rather simple, tautological proposition $\mathsf{Int} \rightarrow \mathsf{Int}$. Of course, since Haskell allows us to write non-terminating programs, every type is inhabited by the non-terminating computation. This means that the Haskell types, viewed as a logical system, is unsound; thus, when encoding a proposition as a Haskell type we should keep in mind that in order to preserve soundness, we must ensure that no non-termination is introduced.

In this section, we focus on one such kind of proposition, that of equality between types. The first key idea of this approach is to encode equality between types as a Haskell type constructor. Figure 4.1 implements an encoding of such an equality. Thus, in Haskell, we can define a type constructor `Equal :* → * → *`, which states that two types are equal

```
data Equal a b = Equal (∀φ. (φ a) → (φ b))

cast :: Equal a b → (φ a) → (φ b)
cast (Equal f) = f
```

This rather elegant trick of encoding the equality between types a and b as a polymorphic function $\forall\varphi.(\varphi\ a) \rightarrow (\varphi\ b)$ has been proposed by Baars and Swierstra [4], and described earlier in a somewhat different setting by Weirich [143]. The logical intuition behind this definition (also known as Leibniz equality [96]) is that two types/propositions are equal if, and only if, they are interchangeable in any context. This context is represented by the arbitrary Haskell type-constructor φ. Another explanation, elaborated in [4], is that since φ is universally quantified, the function with type $\varphi\ a \rightarrow \varphi\ b$ cannot assume anything about the structure of φ, and so the only terminating function with type $\varphi\ a \rightarrow \varphi\ b$ is the identity function.

Given a proof of `(Equal a b)`, we can easily construct functions `a2b::Equal a b → a → b` and `b2a::Equal a b → b → a` which allow us to "cast" between the two types. These casting operations act as elimination constructs on equality types. In addition to casting, we define a number of equality proof combinators that allow us to build new equality proofs from already existing ones.

The general overview with type signatures of these combinators is given Figure 4.1. One can see these combinators as operations on an abstract data-type: more complex equality proofs can be derived from simpler ones algebraically through the use of these combinators.

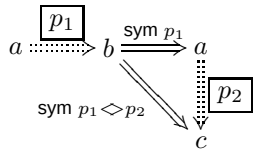
4.2.1 Proof examples

We now give a small example of how equality combinators can be used in constructing new proofs of equality out of old ones. A simple theorem that can be derived about equality can be stated as follows: *For any a, b and c, if $a = b$ then if $a = c$, then $b = c$.*

We can show the proof in natural deduction style. The leaves of the tree are discharged assumptions p_1 and p_2. Using symmetry (for historical reasons called Sym), and then transitivity on the two premises of the root, we derive Equal b c.

$$\dfrac{\dfrac{[p_1 : \mathsf{Equal}\ a\ b]}{\mathsf{Equal}\ b\ a}(\mathsf{Sym}) \quad [p_2 : \mathsf{Equal}\ a\ c]}{\mathsf{Equal}\ b\ c}(\mathsf{Trans})$$

The same proof can be illustrated by a diagram below. The dotted lines (e.g., $a \overset{p_1}{\cdots\!\!\rightarrow} b$) represent the premises, where p_1 :: Equal a b. Equality lines ($b \overset{e}{\Longrightarrow} c$) represent derived equalities, where e :: Equal b c.



How do we prove this theorem in Haskell? As we view Haskell types as propositions, we will first state the above theorem formally as a Haskell type. Under this scheme, the equality $a = b$ becomes the type `(Equal a b)`. Implication is simply the Haskell arrow type.

```
Equals a b → Equals a c → Equals b c
```

Proving this theorem now becomes simply constructing a (terminating) Haskell function that has the above type. We shall call the function `theorem0`, and give its definition below:

```
theorem0       ::  Equals a b → Equals a c → Equals b c
theorem0 p1 p2  =  sym p1 <> p2
```

We now show another proposition and its proof. The proposition is: If $a = b$ and $c = d$, then $(a \rightarrow c) = (b \rightarrow d)$.

In programming, as we will see, the proofs are most frequently used to *cast* between types that can be proved equal. Consider the following example. Suppose that we have a function `f1` of type `a → c`, but we need a function of type `b → d`. Fortunately, we can prove that type `a` equals `b` and `c` equals `d`.

This leads us to state another theorem:

```
Equal a b → Equal c d → (a → c) → (b → d)
```

The proof of this proposition is the function `theorem1` which is defined as follows:

```
theorem1 :: Equal a b → Equal c d →  (a → c)  →  (b → d)
theorem1 p1 p2 f = a2b (subTab p1 p2) f
  -- p1 :: Equal a b
  -- p2 :: Equal c d
  -- subTab p1 p2 :: Equal (a → c) (b → d)
  -- a2b (subTab p1 p2) :: (a → c)  →  (b → d)
```

We start the proof with two premises:

```
p1 :: Equal a b
p2 :: Equal c d
```

Then, we use the combinator (see Figure 4.1)

```
subTab :: Equal a b → Equal c d → Equal (f a c) (f b d)
```

with the premises to obtain the equality proof `Equal (a → c) (b → d)`. The casting operator `a2b` is then used with this combinator to obtain `(a → c) → (b → d)`.

4.2.2 Implementing Equality Proof Combinators

In Figure 4.1 we show a number of functions that manipulate proofs of type equalities. They can roughly be divided into three groups:

1. Proof construction combinators. The types of these combinators correspond to standard properties of equality: reflexivity, symmetry, transitivity and congruence.

2. Casting operators. These functions act as elimination rules for equality. The majority of these operators use the proof that types `a` and `b` are equal to cast from the type `F[a]` to `F[b]`, where `F` is some type context. In the Calculus of Constructions (and similar type theories) this context `F` can be described as a function `F : *->*`, and equality elimination can be given a single type such as `(f:* → *) → (Equal a b) → (f a) → (f b)`. In Haskell, however, we are not allowed to write such arbitrary functions over types, and have to implement a separate combinator for every possible context `F[-]`.

3. Axioms. The axioms allow us to manipulate proofs of equalities of compound types (e.g., pairs) to derive proofs of equalities their constituent parts.

Proof construction

Here, we describe the implementation for each of the combinators that are used to construct equality proofs. We will give definitions of the combinators whose types are listed in Figure 4.1 and comment on the implementation of each one of them. The set of proofs presented below is not complete, even though it seems to be sufficient in practice. New theorems may need to be derived either algebraically by using the existing set of combinators, or, if that proves difficult, by applying the techniques for implementing proof combinators outlined below.

- The simplest of the proof combinators is the reflexivity proof `refl`.

```
data Equal a b = Equal (∀φ. (φ a)→(φ b))
cast :: (Equal a b)→t a→t b
cast (Equal f) = f

-- Algebra for constructing proofs
-- Reflexivity
refl   :: Equal a a
-- Transitivity
trans  :: Equal a b→Equal b c→Equal a c
-- Symmetry
sym    :: Equal a b→Equal b a
-- Congruence
subTa  :: Equal a b→Equal (f a) (f b)
subTab :: Equal a b→Equal c d→Equal (f a c) (f b d)

-- Casting functions
b2a      :: Equal a b→b→a
a2b      :: Equal a b→a→b
castTa   :: Equal a b→c a → c b
castTa_  :: Equal a b→c a d→c b d
castTab  :: Equal a1 a2→Equal b1 b2 →f a1 b1→f a2 b2
castTa__ :: Equal a b→c a d e→c b d e
-- Equality Axioms
pairParts :: Equal (a,b) (c,d)→(Equal a c,Equal b d)
```

Figure 4.1: Representing type equality in Haskell

```
-- reflexivity
refl :: Equal a a
refl = Equal id
```

Although this proof seems trivial, it is often quite useful in programming with equality proofs, as many combinators are derived by manipulating `refl` (see below).

- Transitivity of equality is implemented by the combinator `trans`.

```
-- transitivity
trans :: Equal a b→Equal b c→Equal a c
trans x y = Equal (cast y . cast x)

infixl <>
x <> y = trans x y
```

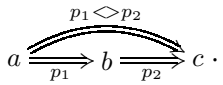
The function trans takes two equality proofs, `x::Equal a b` and `y::Equal b c`, and applies `cast` to them. This results in two functions,

```
cast x :: ∀φ.φ a→φ b
cast y :: ∀φ.φ b→φ c
```

One should note that both of these functions, by definition of `Equal`, must be identity functions (instantiated at their particular types), since they are polymorphic in φ. The composition of these functions yields another function which is polymorphic in type constructor φ (and therefore must be an identity function):

```
cast y . cast x :: ∀φ.φ a→φ c
```

This function can then be wrapped in `Equal` obtaining a proof object of type `Equal a c`.

We shall often write the transitivity combinator as in infix operator (`<>`), taking two equality proofs `p1 :: Equal a b`, and `p2 :: Equal b c`, and producing a proof of `Equal a c`: 

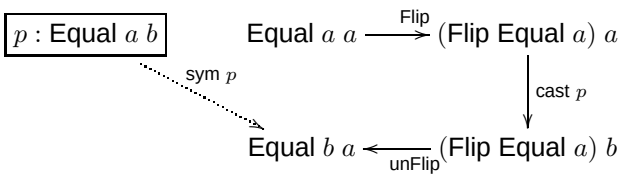
.

- Symmetry is implemented by the combinator `sym`. This combinator has the simple definition we give below, following the development of [4]:

```
newtype Flip f a b = Flip {unFlip :: f b a}

sym :: Equal a b→Equal b a
sym p = unFlip (cast p (Flip refl))
```

The function `sym` implements the proof that equality is symmetric: given a proof that `Equal a b`, it constructs the proof that `Equal b a`. To implement `sym`, we use an auxiliary data-type `Flip`. In function `sym`, we first start with the proof `refl` (that equality is reflexive) which has type `(Equal a a)`. We then apply the constructor `Flip` to `refl` to get a value of type `((Flip Equal a) a)`. Recall that the expression `(cast p)` has the type $\forall\varphi.\varphi\ a \rightarrow \varphi\ b$. In particular, φ can be instantiated to `(Flip Equal a)`. Thus, when `cast p` is applied to `(Flip refl)`, we get a value of type `((Flip Equal a) b)`. Finally, we apply `unFlip` to it to obtain a proof of `Equal b a`. We can illustrate this diagrammatically:



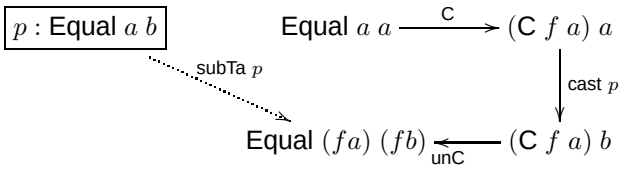
Resorting to this auxiliary data-type (`Flip`) definition is necessary, because Haskell's type system cannot correctly infer the appropriate instantiation of the higher-order type constructor φ.

- The combinator `subTa::Equal a b→Equal (f a) (f b)` (Figure 4.1, line 13) constructs a proof that equality is congruent with respect to application (unary) Haskell type constructors.

```
newtype C f a x = C {unC :: Equal (f a) (f x)}

subTa :: Equal a b -> Equal (f a) (f b)
subTa p = unC (cast p (C refl))
```

We start with the premise `p::Equal a b`. Next, we apply the constructor `C` to a reflexivity proof `refl`, resulting in a value of type `(C f a) a`. The expression `cast p` is applied to this value, obtaining `(C f a ) b`. Finally, `unC` projects a proof of type `Equal (f a) (f b)` from this value. Diagrammatically, this looks as follows:



- The function `subTab` is an instance of congruence of type equality, generalized to binary type constructors.

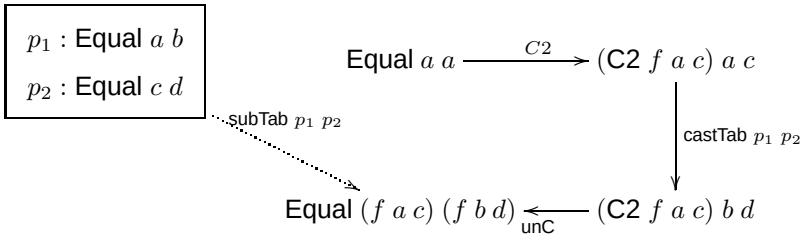
```
newtype C2 f a b x y = C2 { unC2 :: Equal (f a b) (f x y) }

subTab :: Equal a b -> Equal c d -> Equal (f a c) (f b d)
subTab  p1 p2 = unC2 (castTab p1 p2 (C2 refl))
```

It relies on the function `castTab:: Equal a b→Equal c d→(f a c)→(f b d)` (Figure 4.1, line 21) whose definition will be given below. First we obtain an expression of type `(C2 f a b) a b` by applying the constructor `C2` to `refl`. Then, we apply the function `castTab p1 p2` to `C2 refl`. The result has the type `(C2 f a b) c d`. Finally, projecting from `C2` by applying `unC2` produces a proof of the desired proposition `Equal (f a b) (f b d)`.

It is worth noting that `subTab` and `subTa` are very similar. In `subTab`, the auxiliary data-type `C2` plays the same role as the auxiliary data-type `C` in `subTa`. In fact, their definitions are also similar, except that `C2` works on a binary type constructor `f`.

Similarly, `subTa` uses `cast :: Equal a b → f a → f b`, while the definition of `subTab` uses `castTab`, which is merely a generalization of `cast` to a binary type constructor `f`.



Casting operations

Casting operators are elimination rules for equality proofs `Equal`.

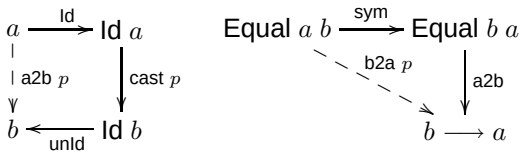
- The simplest of these, `a2b` and `b2a` take proof of type `Equal a b` and return a function that converts from `a` to `b` (and back, respectively).

```
newtype Id x = Id { unId :: x}

a2b :: Equal a b → a → b
a2b p x = unId (cast p (Id x))

b2a :: Equal a b → b → a
b2a = a2b . sym
```

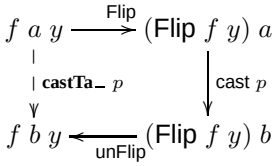
The construction of `a2b` follows a familiar pattern. First, we inject the argument `x` of type `a` into the auxiliary (identity) type constructor `Id`. Then, we apply `cast p` to obtain `Id b`. Finally, we project using `unId` to obtain a `b` object. To define `b2a` we simply invert the proof object and apply `a2b`.



- The function `castTa_` is a form of casting-congruence.

```
castTa_ :: Equal a b→c a d→c b d
castTa_ p x = unFlip (cast p) (Flip x)
```

Starting with a proof `p:Equal a b` and a value `x` of type `f a y`, we first apply `Flip` to `x`, obtaining a value of type `(Flip f y) a`. Then, we apply `cast p`, obtaining `(Flip f y) b`. Finally, we apply `unFlip` to get `f y b`.



- The function `castTab` is simply the composition of functions `castTa_` and `cast`.

```
castTab :: Equal a1 a2 -> Equal b1 b2 -> f a1 b1 -> f a2 b2
castTab p1 p2 = castTa_ p1 . cast p2
```

Starting with an argument of type `f a1 b1`, `castTa_ p1` transforms it into `f a2 b1`. Then, `cast p2` finally returns `f a2 b2`: $f\ a_1\ b_1 \xrightarrow{\text{castTa\_}\ p_1} (f\ a_2)\ b_1 \xrightarrow{\text{cast}\ p_2} (f\ a_2)\ b_2$.

- The function `castTa__` further generalizes casting to ternary type constructors.

```
newtype Flip3 f a b c = Flip3 {unFlip3 :: f b c a}
castTa__ :: Equal a b -> c a x y -> c b x y
castTa__ p x = unFlip3 (castTa_ p (Flip3 x))
```

A Note On Strategies for Implementing Equality Operations

From the previous examples, we can observe a more general pattern of programming with equality proofs and deriving equality combinators. Usually, one starts with some equality proof `p` of type `Equal a b`, and the goal is to produce a function that transforms some other type (`R[a]`) to type `R[b]`.[1]

For example, if we have a type `Int→Bool` and an equality type `Equal c Bool`, one should be able to derive the type `Int→c` from it. Haskell's type checker, however, is not designed to make this conclusion automatically. Instead, the programmer must devise a type constructor `R`, so that applying `R` to `Bool` produces the type `(Int→Bool)` and applying `R` to `c` to produce `(Int→c)`. Then, casting

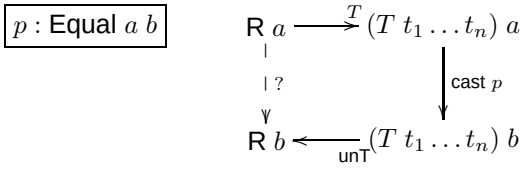
[1] The can be thought of as a type `R` context with one hole.

operations on (i.e., `castTa`) that type constructor allow the programmer to derive `R c`, from `R Bool`, which is the same as `Int` $\rightarrow$ `c`.

More generally, the implementation of many equality combinators usually proceeds as follows: the first step is to take apart the proof p using the `cast` combinator to obtain a function f of type $\forall\varphi.\, \varphi\ \mathsf{a} \rightarrow \varphi\ \mathsf{b}$. Note that the polymorphic variable φ can then be instantiated to any unary type constructor. The next step is to define an auxiliary $(n+1)$-ary type constructor `T`.

```
data T t_1 ... t_n x = T (R x)
unT :: T t_1...t_n x -> R x
```

The type constructor `T` is a function of the context `R` in which we want to substitute `b` for `a`. Then, a value of type $(T\ t_1 \ldots t_n)\ a$ is created. When the function f is applied to this value, φ becomes instantiated to $T\ t_1 \ldots t_n$, and the result is of type $(T\ t_1 \ldots t_n)\ b$. Finally, the function `unT` is used to project the desired final result in which `a` has been replaced by `b`. We can show this pattern diagrammatically:



Axioms

A number of "equality axioms" are also postulated. The axiom most commonly used in the examples that follow is `pairParts`:

```
pairParts :: Equal (a,b) (c,d) → (Equal a c, Equal b d)
pairParts = -- primitive
```

These axioms specify how Haskell type constructors (e.g., products, sums, and so on) behave with respect to equality. The `pairParts` axiom allows us to conclude that if two products are equal, then so are their constituent parts.

It has been argued that axioms such as `pairParts` cannot be defined in Haskell itself [19]. In our framework, they are introduced as primitive constants. We conjecture that this does not compromise the consistency of the overall Haskell type system, but we can offer no proof at this time. In practical terms, one possible implementation[2] of `pairParts` would be

[2] Here `unsafeCast` is the function with the type $\mathsf{a} \rightarrow \mathsf{b}$. Strictly speaking, this function should not exist in standard Haskell, but it can be written in most Haskell implementations using a well-known "unsafe IO reference trick."

```
pairParts :: Equal (a,b) (c,d)→(Equal a c, Equal b d)
pairParts (Equal f) = (Equal (unsafeCast f), Equal (unsafeCast f))
```

4.3 Proofs and Runtime Representations in Haskell

How do we use equality proofs? In this section, we will illustrate this by developing a small implementation of operations on natural numbers. Natural numbers are defined inductively as the least set closed under the rules:

$$\frac{}{\mathsf{Z} \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{\mathsf{S}\ n \in \mathbb{N}}$$

The type of natural numbers in a functional language can be thought of as a logical proposition. This type is inhabited by an infinite number of distinct proofs, each of which can be identified with a particular natural number. For example, the Haskell data-type `Nat` is such a type:

```
data Nat = Zero | Succ Nat
```

Note that each expressions of type `Nat` is an equally valid proof of this proposition. For example, `Zero : Nat`, but also `Succ Zero : Nat`.

As we have seen, individual natural numbers cannot be distinguished from one another at the level of types. There are, however, interesting properties of individual natural numbers that can be useful in types. For example, we might want to know that the type of an array indexing function takes an index which is provably less than the size of the array. If this property can be specified and proved statically in the type system, then we can dispense with runtime array bounds checking without compromising safety of the program.

To make assertions about particular natural numbers in types, e.g., asserting their equality, we need first to represent natural numbers *at the level of types*, where each natural number corresponds to some type. Thus, we define the following two data-types

```
data Z = Z
newtype S x = S x
```

The type `Z` has one constructor, also named `Z`, and represents the natural number zero. The successor operation is represent by the type constructor `S : * → *`. The intended meaning is that the expression `S (S Z) : S (S Z)` represents the natural number `2` at the type level.

One should note at this point, that the two types `Z` and `S` are not by themselves enough to encode natural numbers at type level. In fact, we could refer to them as *pre-numbers*: one could apply type constructor `S`

to any Haskell type. Thus, `S "X":(S String)` clearly does not represent a natural number. How can we enforce the requirement that the naturals are well-formed?

The solution is to use equality types to define a Haskell type constructor corresponding to the inductive judgment $n \in \mathbb{N}$.

```
data IsNat n = IsZero (Equal n Z)
             | ∀α. IsSucc (IsNat α)
                    (Equal n (S α))
```

$$\left| \quad \frac{}{\mathsf{z} \in \mathbb{N}} \quad \frac{n \in \mathbb{N}}{\mathsf{S}\, n \in \mathbb{N}} \right.$$

The data-type `IsNat :: * → *` is just such a type constructor: we read `IsNat n` as $n \in \mathbb{N}$. In defining the data-type `IsNat`, we define a data-constructor for every derivation rule of the inductive judgment $n \in \mathbb{N}$. Thus, inhabitants of `IsNat n` act as proofs that $n \in \mathbb{N}$: for every derivation of the judgment $n \in \mathbb{N}$, there is a value of type `IsNat n`[3].

1. The constructor `IsZero` implements the base case of the proof of the judgment $n \in \mathbb{N}$, $\frac{}{\mathsf{z} \in \mathbb{N}}$. It takes as its single argument a proof that the argument type is equal to `Z`.

2. The constructor `IsSucc` is the inductive step $\frac{n \in \mathbb{N}}{\mathsf{s}\, n \in \mathbb{N}}$: as its first argument it takes the proof of the antecedent judgment `IsNat` α. Its second argument is the proof that `n` equals to the successor of this α, where α is some existentially quantified type representing a natural number.

Smart Constructors. Recall that the type constructor `IsNat` has one argument, a type representing the natural number `n`, such that `IsNat n` means that $n \in \mathbb{N}$. We shall call this type argument the *index* of `IsNat`.

In the definition of the inductive judgment $n \in \mathbb{N}$, we use pattern matching to specify the shape of the index. For example, the base case rule forces the index to be zero: $\frac{}{\mathsf{z} \in \mathbb{N}}$. In Haskell data-types, however, we cannot pattern match on the index. Rather, we use equality proofs as additional premises to force a particular "shape" on the index type argument.

Hence, Haskell gives the constructor `IsZero` the type `Equal n Z → IsNat n`. When `IsZero` is applied to `refl::Equal a a`, the type variable `a` is unified with `Z`, obtaining the typing:

```
IsNat refl :: IsNat Z
```

This pattern is captured by the value `z` and function `s` whose definition is given below:

[3] With the usual caveat that such values do not contain non-terminating computations.

```
z :: IsNat Z
z = IsZero refl

s :: IsNat n → IsNat (S n)
s n = IsSucc n refl
```

The functions `z` and `s` are called "smart constructors," since they correspond to the data constructors of `IsNat`, but also do some useful work. Note that the type of `z::IsNat Z` corresponds now exactly to the judgment form $\frac{}{\mathsf{z} \in \mathbb{N}}$.

Similarly, the constructor `IsSucc` has the type `IsNat m → Equal b (S n) → IsNat b`. The smart constructor `s` takes an argument of type `IsNat n`. Then, it applies the constructor `IsSucc` to it, obtaining

```
IsSucc n :: Equal b (S n) → IsNat b
```

Finally, the resulting function is applied to `refl`. This forces the type variable `b` to be unified with `S n`, obtaining the result of type `S n`.

Note the use of existential types in defining constructors. Existential quantification and equality do not appear in the rule $\frac{n \in \mathbb{N}}{\mathsf{s}\ n \in \mathbb{N}}$. However, the type of the smart constructor `s::IsNat n → IsNat (S n)` again directly corresponds to the judgment $\frac{n \in \mathbb{N}}{\mathsf{s}\ n \in \mathbb{N}}$.

Runtime values. Another thing to note is that there is a one-to-one correspondence between natural numbers (at the value level) and elements of the data-type `IsNat`. The isomorphism is easily constructed by induction over natural numbers and judgments of $n \in \mathbb{N}$. For example, the expression `s z : IsNat (S Z)` is the only (if we ignore the bottom element in Haskell semantics) element of the type `IsNat (S Z)`. This property is quite useful, since it implies that we can use the values of type IsNat n to represent individual natural numbers as *runtime values*, as well as proofs that a particular type `n` is a representation of a natural number.

The `IsNat` type also bears a strong resemblance to the notion of *singleton types* [116, 58]. In the FLINT [114, 116] compiler framework, a data-type for natural numbers (for example) is represented *at the level of kinds*, as an inductive kind `Nat`. This kind classifies a set of *types* $\{0,1,2,\ldots\}$. However, there is also a type `snat : Nat → *`, which classifies *runtime* natural numbers . Each runtime natural number value $\hat{n}$ has the type `snat n`. The typing rules in such a system might look like:

$$\frac{\Delta \vdash n : \mathsf{Nat}}{\Delta, \Gamma \vdash \hat{n} : (\mathsf{snat}\ n)}(\text{Literal}) \qquad \frac{\Delta, \Gamma \vdash e_1 : \mathsf{snat}\ m \quad \Delta, \Gamma \vdash e_2 : \mathsf{snat}\ n}{\Delta, \Gamma \vdash e_1 \hat{+} e_2 : (\mathsf{snat}(m+n))}(\text{Plus})$$

In our implementation, type constructors `S` and `Z` play the role of natural numbers at the type level; the type constructor `IsNat` plays the role of `snat`, values of types `IsNat Z`, `IsNat (S Z)`, and so on, play the role of runtime naturals. The only difference with FLINT is that there is no way to represent the *inductive kind* `Nat` itself – the well-formedness of naturals at the type level must be enforced by the inductive definition of `IsNat`.

Predicates

`IsOdd` and `IsEven` are two mutually inductive predicates on natural numbers, defined as the least relations that satisfy the rules:

$$\frac{}{\mathsf{IsEven\ z}} \qquad \frac{\mathsf{IsOdd}\ n}{\mathsf{IsEven}\ (\mathsf{s}\ n)} \qquad \frac{\mathsf{IsEven}\ n}{\mathsf{IsOdd}\ (\mathsf{s}\ n)}$$

Here, we will show how those predicates can be encoded using equality types in Haskell.[4] First, for clarification, let us tentatively assign a "type" to these predicates. In a dependently typed system such as Coq [43, for the same example], predicates `IsOdd` and `IsEven` would be given a type:

```
IsOdd, IsEven : (n : nat) → Prop
```

In our Haskell encoding, we collapse this distinction: both naturals and propositions are types of kind *. Thus, we define two type constructors `IsEven` and `IsEven` which have the kind * → *:

```
data IsEven t =      Z_Even  (Equal t Z)
              | ∀n.  S_Even  (Odd n)  (Equal t (S n))

data IsOdd t =  ∀n.  S_Odd   (Even n) (Equal t (S n))
```

We also define the corresponding "smart constructors" which allow us to easily build proofs of these predicates:

```
z_even :: Even Z
z_even = Z_Even refl

s_even :: IsOdd n → IsEven (S n)
s_even x = S_Even x refl
```

[4] While we shall refer to the language as "Haskell," it is important to remember that we use more features than available in Haskell 98 [64] (higher rank polymorphism, existential types, and so on). All of these features are available in the most popular Haskell implementations.

```
s_odd :: IsEven n → IsOdd (S n)
s_odd x = S_Odd x refl
```

The first example we present is the function `oddOrEven`. This function proves the property of natural numbers that $\forall n \in \mathbb{N}.\mathsf{IsEven}\ n \vee \mathsf{IsOdd}\ n$.

The disjunction of two propositions is represented using Haskell's `Either` data-type:

```
data Either a b = Left a | Right b
```

In the implementation below, we will represent the proposition $(\mathsf{IsEven}\ n) \ \vee \ (\mathsf{IsOdd}\ n)$ by auxiliary data-type `IsOddOrEven::* → *`.

This is not strictly necessary, but it enables us to express the desired property of being odd or even as an application of a unary type constructor. This, in turn, makes the implementation less verbose, since the equality proof and casting combinators are more concise when working with unary constructors. We examine the function `oddOrEven` in more detail:

```
1  newtype IsOddOrEven n = OE (Either (IsOdd n) (IsEven n))
2
3  l = OE . Left
4  r = OE . Right
5
6  oddOrEven :: IsNat n → IsOddOrEven n
7  oddOrEven (IsZero p) = castTa (sym p) (r z_even)
8  oddOrEven (IsSucc n p) =
9    case oddOrEven n of
10      OE (Left op) → castTa (sym p) (r (s_even op))
11      OE (Right op → castTa (sym p) (l (s_odd op))
```

Line 7 is the base case of this function:

```
oddOrEven (IsZero p) = castTa (sym p) (r z_even)
 -- p                          :: Equal n Z
 -- r z_even                   :: IsOddOrEven Z
 -- castTa (sym p) (r z_even)  :: IsOddOrEven n
```

If a runtime representation of the natural number `Z` is given, then we construct a base case for

even number: the expression `(r z_even)` has type `(IsOddOrEven Z)`; then, the proof `(sym p) ::Equal Z n` is then used to cast back to `(IsOddOrEven n)`.

Similarly, in the inductive step (lines 10 and 11), first construct a proof recursively, and then, depending on whether the recursive proof is odd or even, construct the next even or odd proof, respectively.

```
oddOrEven (IsSucc n p) =
  case oddOrEven n of
     OE (Left op)→ castTa (sym p) (r (s_even op))
     OE (Right op)→ castTa (sym p) (l (s_odd op))
```

A similar and important function is one that constructs the proof of equality between two naturals. This function is an instance of a common pattern in programming with equality proofs: two values whose types are judgments indexed by types a and b are compared structurally to possibly obtain a result of type `Equal a b` (hence the `Maybe` type in the range of `equalNat`). This is, in effect, a runtime check which allows us to convert between types a and b.

```
equalNats :: IsNat a→IsNat b→Maybe (Equal a b)
equalNats (IsZero p1) (IsZero p2) = return ( p1 <> (sym p2))
      -- p1             :: Equal a Z
      -- p2             :: Equal b Z
      -- p1 <> (sym p2) :: Equal a b
equalNats (IsSucc n1 p1) (IsSucc n2 p2) =
    do { p3 <- equalNats n1 n2
       ; return (p1 <>  (subTa p3) <> (sym p2) }
      -- p1       :: Equal a (S _1)
      -- p2       :: Equal b (S _2)
      -- p3       :: Equal _1 _2
      -- subTa p3 :: (S _1) (S _2)
      -- p1 <> (subTa p3) <> (sym p2) :: Equal a b
equalNats _ _ = Nothing
```

Example: Arithmetic

As our next example we implement addition of natural numbers. The addition function in the encoding of natural numbers presented above has the following properties: it takes two arguments, integers n and m, and returns an integer z, such that $z = n + m$. So, what type do we give our function in Haskell?

```
plus :: IsNat a -> IsNat b -> IsNat (? a b)
```

The only valid thing we can give in place of the question mark would be a type function of kind $* \rightarrow * \rightarrow *$. However, such functions[5] are not permitted by type systems of most practical programming languages including Haskell. Thus, we must encode addition at the type level indirectly. First, although we do not have computation and functions at type level, we *can* use type constructors to simulate relations between types. Thus, we define addition as an inductive relation PlusRel $m\ n\ i$, where $i = m + n$.

$$\frac{}{\mathsf{PlusRel}\ \mathsf{z}\ m\ m} \quad \exists i \in \mathbb{N}.\ \frac{\mathsf{PlusRel}\ n\ m\ i}{\mathsf{PlusRel}\ (\mathsf{s}\ n)\ m\ (\mathsf{s}\ i)}$$

We encode this relation as a ternary Haskell type constructor

```
1  --  PlusRel :: * → * → * → *
2  data PlusRel m n i =
3      Z_PlusRel   (Equal m Z) (Equal n i)
4   |  ∀αβ. S_PlusRel (PlusRel β n α) (Equal m (S β)) (Equal i (S α))
5
6  zPlusRel :: PlusRel Z i i
7  zPlusRel = Z_PlusRel refl refl
8
9  sPlusRel :: PlusRel m n i→ PlusRel (S m) n (S i)
10 sPlusRel p  = S_PlusRel p refl refl
```

Now, we are ready to define the addition function. There are two steps to creating this function. The first, intermediate step is the function `pl`.

```
pl :: IsNat m → IsNat n → PlusRel m n z → IsNat z
pl _ n r = p n r where
     p :: IsNat n → PlusRel m n z → IsNat z
     p  n (RPZ p1 p2) = (cast p2 n)
     p  n (RPS r p1 p2 ) =  cast (sym p2) (s (p n r))
```

This function takes three arguments: two natural numbers m and n, and a proof that $m + n = z$. It is defined in terms of the function `p`, which is defined inductively on the structure of the proof of addition relation `PlusRel m n z`: from `PlusRel m n z`, and the representation of `n`, `p` is able to construct the proof of the judgment `IsNat z`. In computational terms, this is equivalent to constructing the natural number representing the resulting sum. Of course, this function is not all that useful since it requires the

[5] As opposed to *type constructors*.

proof of the judgment `PlusRel` as one of its arguments. It is possible to construct this proof out of $m \in \mathbb{N}$ and $n \in \mathbb{N}$.

What would the type of such a function look like? One possibility is

```
constructProof :: IsNat m → IsNat n → PlusRel m n z
```

However, the type variable `z` appears only on the positive side of the arrow type above which would mean that a `PlusRel m n z` can be constructed for *all* types `z`. This is patently false. The problem is that of quantification: given any two natural numbers m and n, we can construct the proof that for *some* z, $m + n = z$. Thus, we need to existentially quantify the type variable `z`. The type of `constructProof` would then look like:

```
constructProof :: IsNat m → IsNat n → ∃α.PlusRel m n α
```

The function `plus` defined below (lines 5-11) performs the actions of `constructProof` and `pl` simultaneously, yielding a result of type $\exists\alpha.(\mathsf{PlusRel}\ m\ n\ \alpha) \times (\mathsf{IsNat}\ \alpha)$. Slightly complicating the notation below is the fact that in Haskell existential types can only be used in data-type definitions. Therefore, we define an auxiliary data-type `Exists`. This type constructor takes a unary type constructor `f` and implements the existential type $\exists\alpha.\mathsf{f}\ \alpha$.

```
1  --   Exists φ ≡ ∃α.φ(α)
2  data Exists f = ∀α. Exists (f α)
3  data Pl x y z = Pl (PlusRel x y z) (IsNat z)
4
5  plus :: (IsNat x) → (IsNat y) → (Exists (Pl x y))
6  plus (IsZero p1) m =
7     Exists (((Pl (castTa__ (sym p1) (zPlusRel)) m)))
8  plus (IsSucc n p1) m =
9    case plus n m of
10     Exists (Pl pj y) →
11       Exists  ((Pl (castTa__ (sym p1) (sPlusRel pj)) (s y)))
```

Example: Putting `IsNat` into the `Num` Class

In the `IsNat` encoding, each natural number has a different (and incompatible) type: the number one has the type `IsNat (S Z)`, the number two has the type `IsNat (S (S Z))`, and so on. Is there a type that represents the entire set of natural numbers? Naturally, there *is* such a type and it is $\exists\alpha.\ \mathsf{IsNat}\ \alpha$. Thus, we can finally implement a traditional Haskell addition function by declaring `(Exists IsNat)` to be an instance of the class `Num`.

```
instance Num (Exists IsNat) where
  (+) (Exists m) (Exists n) =
      let Exists (Pl prf z) = plus m n
      in  Exists z
```

Example: Encoding the Ordering Relation

Another interesting relation on natural numbers is ordering. The relation $m \leq n$ on natural numbers can be defined by induction on n as the least relation that satisfies the rules

$$\frac{}{n \leq n}(\texttt{LEQ-Refl}) \quad \frac{m \leq n}{m \leq \texttt{S}\, n}(\texttt{LEQ-Succ})$$

.

The implementation in Haskell consists of the data-type `LEQ::*` $\rightarrow$ `*` $\rightarrow$ `*` and the corresponding pair of smart constructors.

```
data LEQ m n =
         LEQ_Refl (Equal m n)
   | ∀α. LEQ_S (LEQ m α) (Equal n (S α))

leq_refl :: LEQ a a
leq_refl = LEQ_Refl refl

leq_s :: LEQ a b -> LEQ a (S b)
leq_s s = LEQ_S s refl
```

With the Haskell implementation of $\leq$, we can begin to construct interesting proofs.

```
compLEQ :: (IsNat m)→(IsNat n)→Maybe (LEQ m n)
compLEQ (IsZero p1) (IsZero p2) = return (LEQ_Refl (p1 <> (sym p2)))
compLEQ (z@(IsZero p1)) (IsSucc n' p2) =
 do { r←compLEQ z n'; return (LEQ_S r p2) }
compLEQ (IsSucc m p1) (IsZero p2) = Nothing
compLEQ (IsSucc m p1) (IsSucc n p2) =
 do { r←compLEQ m n; return (castTab (sym p1) (sym p2) (theorem1 r)) }

newtype Th1 x y = Th1 {unTh1 :: LEQ (S x) (S y)}

theorem1 :: (LEQ m n) → (LEQ (S m) (S n))
theorem1 (LEQ_Refl p1) = unTh1 (castTa_ (sym p1) (Th1 leq_refl))
```

The function `compLEQ`, presented above, takes two natural number representations (of types `IsNat m` and `IsNat n`), and returns the proof that $m \leq n$, if such a proof can be constructed. We will examine this function more closely to familiarize ourselves with the practice of programming with these encodings. The construction of `LEQ m n` proceeds by induction on the structure of the two numbers.

The base case (line 2) assumes that both numbers are zero.

```
(2) compLEQ (Zero p1) (Zero p2) = return  (LEQ_Refl (p1 <> (sym p2)))
```

The proofs `p1` and `p2` have types `Equal m Z` and `Equal n Z`, respectively. The combined proof `p1 <>(sym p2)` has the type `Equal m n`. This is exactly what the base case constructor for `LEQ` requires, and is used to build the proof that `LEQ m n`.

The second case (lines 3-4) is the case when the first argument is zero, and the second is some `Succ n'`.

```
(3) compLEQ (z@(Zero p1)) (Succ n' p2) =
     do { r ← compLEQ z n'; return (LEQ_S r p2) }
```

This proceeds by constructing the proof `compLEQ z n'` of the type `LEQ Z n'`. Then, the proof `p2 :: Equal n' (S n)` is used to construct `LEQ Z n`.

The following case always returns `Nothing`, since no non-zero nat is less than zero.

```
compLEQ (Succ m p1) (Zero p2) = Nothing
```

Finally, the inductive step where both numbers are non-zero (lines 6-7) is the most interesting one:

```
(6) compLEQ (Succ m p1) (Succ n p2) =
      do { r ← compLEQ m n
         ; return (castTab (sym p1) (sym p2) (theorem1 r)) }
      -- m          :: IsNat _1
      -- n          :: IsNat _2
      -- p1         :: Equal m (S _1)
      -- p2         :: Equal n (S _2)
      -- sym p2     :: Equal (S _2) n
      -- r          :: LEQ _1 _2
      -- theorem1 r :: LEQ (S _1) (S _2)
```

The two arguments are taken apart and variables `m` and `n` have types `m :: IsNat _1` and `n :: IsNat _2`. [6] There are also two proofs, `p1 :: Equal m (S _1)` and `Equal n (S _2)`. The recursive call to `compLEQ s1 s2` produces an inequality proof of type `LEQ _1 _2`, and the function

[6] We use the notation _1, and so on to indicate types of Skolem constants in Haskell type checking of existential type eliminations.

```
theorem1 :: LEQ m n -> LEQ (S m) (S n)
```

is used to obtain the proof of type `LEQ (S _1) (S _2)`. Finally, proofs `m` and `n` are used to cast back to type `LEQ m n`.

Chapter 5

Language Implementation Using Haskell Proofs

In Section 4.3 we have familiarized ourselves with basic techniques of encoding judgments and their proofs in Haskell, and with programming using these proofs. Next, we introduce our first heterogeneous meta-programming example utilizing these techniques. This development proceeds in a number of steps: first, we define an object language; then, we introduce a runtime representation of types of the object language(Section 5.1); then we introduce an encoding of *well-typed terms* for the object language defined in Section 5.2. The implementation consists of a type of object language typing judgments, an interpreter that evaluates the proofs of those judgments to meta-language values, and a type-checker that constructs typing judgment proofs.

5.0.1 The Language L_1

First, we present is the language L_1. The language L_1 (Figure 5.1) is a small, simply typed functional language. We explain the relevant definitions in some more detail before proceeding onto the implementation of L_1.

Syntax of L_1. The syntax of L_1 consists of three inductively defined sets.

$$\begin{aligned}
\tau \in \mathbb{T} ::= &\quad \mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
\Gamma \in \mathbb{G} ::= &\quad \langle\rangle \mid \Gamma, \tau \\
e \in \mathbb{E} ::= &\quad n \mid \lambda\tau.e \mid e_1\ e_2 \mid \mathsf{Var}\ n \mid e_1 \oplus e_2 \mid (e_1, e_2) \mid \pi_{\{0,1\}}\ e
\end{aligned}$$

First, there is a set of types, τ, which includes natural numbers (or some other base types), function spaces ($\tau_1 \to \tau_2$), and binary products ($\tau_1 \times \tau_2$). Second, there is a set of type assignments, Γ, which are sequences of types. The type τ in a type assignment of Γ at position n assigns type τ to the free variable $\mathsf{Var}\ n$. Third, there is a set of expressions which contain the usual lambda calculus constructs presented in Church style (domains of abstractions are explicitly typed). Variable binding is expressed in the positional

style, counting the number of intervening binding sites prior to the binding of the variable itself [13, 14]. Support for integer literals, and arithmetic operators ($e_1 \oplus e_2$) is also included.

Static semantics. The type system of L_1 is also shown in Figure 5.1: the presentation is that of a small applied simply typed λ-calculus.

$$\frac{}{\Gamma \vdash n : \mathsf{int}}(\text{Lit}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}}(\text{Arith}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau \vdash \mathsf{Var}\ n : \tau}(\text{Var})$$

$$\frac{\Gamma, \tau \vdash e : \tau'}{\Gamma \vdash \lambda\tau.e : \tau \to \tau'}(\text{Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}(\text{App})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}(\text{Pair}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash (\pi_1\ e) : \tau_1}(\text{Pi1}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash (\pi_2\ e) : \tau_2}(\text{Pi2})$$

$$\frac{}{\Gamma, \tau \vdash 0 : \tau}(\text{VarZ}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau' \vdash (n+1) : \tau}(\text{VarS})$$

The typing judgment is fairly standard. It is defined by structural induction on L_1 expressions. Type assignments grow when they encounter the λ-abstraction. When a free variable $\mathsf{Var}\ n$ is encountered, an auxiliary judgment $\Gamma \vdash n : \tau$ is used (rules **VarS** and **VarZ**). This judgment is defined by induction over the variable index: if the variable index is greater then zero, this judgment weakens the context and decrements the index until the **VarZ** rule applies.

Semantics of L_1. The semantics of L_1 presented in Figure 5.1 are given in the denotational style [127, 50]. The semantic functions are set-theoretic maps from syntactic sets to the corresponding semantic sets. There are three such maps:

1. First, types are mapped into semantic sets. The type of naturals is mapped to the set of natural numbers. Arrow types are mapped into function spaces, product types into products of underlying sets.

$$\begin{aligned} [\![\bullet]\!] &: \tau \to \mathbf{Set} \\ [\![\mathsf{int}]\!] &= \mathbb{N} \\ [\![\tau_1 \to \tau_2]\!] &= ([\![\tau_1]\!] \to [\![\tau_2]\!]) \\ [\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \end{aligned}$$

2. The semantics of type assignments is a "nested" product of the types in the assignment:

$$\begin{aligned}
[\![\bullet]\!] &: \Gamma \to \mathbf{Set} \\
[\![\langle\rangle]\!] &= \mathbf{1} \\
[\![\Gamma, \tau]\!] &= [\![\Gamma]\!] \times [\![\tau]\!]
\end{aligned}$$

3. Finally, the semantics of L_1 programs is defined in "categorical style," by induction over the typing derivations of L_1. The semantic function $[\![\bullet]\!]$[1] takes a proof of a judgment $\Gamma \vdash e : \tau$ and produces "an arrow," i.e., a function from the meaning of the type assignments to the meaning of the type of the expression e. In its variable case, the semantic function relies on the auxiliary family of semantic functions $\mathcal{L}[\![\bullet]\!] : (\Gamma \vdash n : \tau) \to ([\![\Gamma]\!] \to [\![\tau]\!])$, which for some integer n, performs the look-up of the n-the element of the runtime environment:

$$\begin{aligned}
\mathcal{L}[\![\bullet]\!] &: (\Gamma \vdash n : \tau) \to ([\![\Gamma]\!] \to [\![\tau]\!]) \\
\mathcal{L}[\![0]\!]\ (_, v) &= v \\
\mathcal{L}[\![n+1]\!]\ (\rho, _) &= \mathcal{L}[\![n]\!]\ \rho
\end{aligned}$$

The semantic function is defined as follows:

$$\begin{aligned}
[\![\bullet]\!] &: (\Gamma \vdash e : \tau) \to ([\![\Gamma]\!] \to [\![\tau]\!]) \\
[\![\Gamma \vdash \mathsf{Var}\ n]\!]\ \rho &= \mathcal{L}[\![n]\!]\ \rho \\
[\![\Gamma \vdash \lambda\tau_1.e : \tau_1 \to \tau_2]\!]\ \rho &= (x : [\![\tau_1]\!]) \mapsto ([\![\Gamma, \tau_1 \vdash e : \tau_2]\!]\ (\rho, x)) \\
[\![\Gamma \vdash e_1\ e_2 : \tau]\!]\ \rho &= [\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\ \rho\ ([\![\Gamma \vdash e_2 : \tau']\!]\ \rho) \\
[\![\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2]\!]\ \rho &= ([\![\Gamma \vdash e_1 : \tau_1]\!], [\![\Gamma \vdash e_2 : \tau_2]\!]) \\
[\![\Gamma \vdash \pi_n\ e : \tau_n]\!]\ \rho &= \pi_n\ [\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]
\end{aligned}$$

Basic Properties of L_1. For the sake of completeness of our presentation, we state some basic properties of the language L_1. These are fairly standard (e.g., [5]), but will be useful in justifying some design choices in the latter implementation of L_1.

Proposition 2 (Generation lemma for L_1) *The following implications hold:*

1. $\Gamma \vdash \mathsf{Var}\ n : \tau \Rightarrow \Gamma \vdash n : \tau$

[1] One should note that the semantic function $[\![\bullet]\!]$ is actually a *family* of functions indexed by e,Γ and τ, and as such is given a dependent type

$$\Pi_{e \in E, \Gamma \in \mathbf{\Gamma}, \tau \in T} (\Gamma \vdash e : \tau) \to [\![\Gamma]\!] \to [\![\tau]\!]$$

2. $\Gamma \vdash e_1\ e_2 : \tau \Rightarrow \exists \tau'.\Gamma \vdash e_1 : \tau' \to \tau$ *and* $\Gamma \vdash e_2 : \tau'$
3. $\Gamma \vdash (\lambda \tau_1.e) : \tau \Rightarrow \exists \tau_2.\tau = \tau_1 \to \tau_2$ *and* $\Gamma, \tau_1 \vdash e : \tau_2$.
4. $\Gamma \vdash (e_1, e_2) : \tau \Rightarrow \exists \tau_1, \tau_2.\tau = \tau_1 \times \tau_2$ *and* $\Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2$
5. $\Gamma \vdash \pi_1\ e : \tau \Rightarrow \exists \tau'.\Gamma \vdash e : \tau \times \tau'$
6. $\Gamma \vdash \pi_1\ e : \tau \Rightarrow \exists \tau'.\Gamma \vdash e : \tau' \times \tau$
7. $\Gamma, \tau \vdash 0 : \tau$
8. $\Gamma, \tau' \vdash (n+1) : \tau \Rightarrow \Gamma \vdash n : \tau$

PROOF. Proof is by induction on the height of derivations, as in [5]. □

Proposition 3 (Uniqueness of derivations) *For all $e \in E, \tau \in T, \Gamma \in \Gamma$, if $\Gamma \vdash e : \tau$, then there is only one derivation tree that is the proof of $\Gamma \vdash e : \tau$.*

PROOF. Proof is by induction on the height of derivation $\Gamma \vdash e : \tau$ and using the generation lemma. □

5.0.2 Implementation of L_1: an Overview

The implementation of L_1 in many ways mirrors the definitions in Section 5.0.1, in so far as it, too, L_1 consists of three "artifacts." One could view the three artifacts as *syntax, semantics* and *pragmatics* of the language L_1, respectively:

1. A data-type representing typing judgments of L_1. The inhabitants of this type represent typing derivations of L_1. This data-type, which we will call (well-typedness) *judgments*, is similar to the inductively defined types and relations from Section 4.3.
2. An interpreter which defined over proofs of typing judgments of L_1. The interpreter is a (total) a mapping from well-typed judgments to the meanings of types for those judgments, and thus directly corresponds to the family of semantic functions $[\![\bullet]\!]$.
3. A type-checking function. This function takes syntactic (not necessarily well-typed) L_1 pre-terms and constructs a proof of L_1 typing judgment or raises an error. This function has no direct correspondence to the semantic definitions given from (Figure 5.1). Rather, it implements a well-formedness condition on L_1 typing derivations that is assumed implicitly by those definitions.

Syntax

Types	$\tau ::=$	$\mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$
Assignments	$\Gamma ::=$	$\langle\rangle \mid \Gamma, \tau$
Expressions	$e ::=$	$n \mid \lambda\tau.e \mid e_1\ e_2 \mid \mathsf{Var}\ n \mid e_1 \oplus e_2$ $(e_1, e_2) \mid \pi_{\{0,1\}}\ e$

Type System

$$\frac{}{\Gamma \vdash \mathsf{Lit}\ n : \mathsf{int}} \qquad \frac{\Gamma \vdash n : \tau}{\Gamma \vdash \mathsf{Var}\ n : \tau}$$

$$\frac{\Gamma, \tau \vdash e : \tau'}{\Gamma \vdash \lambda\tau.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau'\ \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{}{\Gamma, \tau \vdash 0 : \tau} \qquad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau' \vdash (n+1) : \tau}$$

SEMANTICS

Assignments

$$[\![\langle\rangle]\!] = ()$$
$$[\![\Gamma, \tau]\!] = ([\![\Gamma]\!], [\![\tau]\!])$$

Types

$$[\![int]\!] = \mathsf{Int}$$
$$[\![t1 \to t_2]\!] = [\![t1]\!] \to [\![t2]\!]$$
$$[\![(t1, t2)]\!] = ([\![t1]\!], [\![t2]\!])$$

Judgments

$$[\![\bullet]\!] : (\Gamma \vdash e : \tau) \to [\![\Gamma]\!] \to [\![\tau]\!]$$
$$[\![\Gamma \vdash \mathsf{Var}\ n : \tau]\!]\ \rho = \mathcal{L}[\![n]\!]\ \rho$$
$$[\![\Gamma \vdash \lambda\tau.e : \tau \to \tau']\!]\ \rho = (x : [\![\tau]\!]) \mapsto ([\![\Gamma, \tau \vdash e : \tau']\!]\ (\rho, x))$$
$$[\![\Gamma \vdash e_1\ e_2 : \tau']\!]\ \rho = ([\![\Gamma \vdash e_1 : \tau \to \tau']\!]\ \rho)([\![\Gamma \vdash e_2 : \tau]\!]\ \rho)$$
$$\mathcal{L}[\![0]\!]\ (\_, v) = v$$
$$\mathcal{L}[\![n+1]\!]\ (\rho, \_) = \mathcal{L}[\![n]\!]^V\ n$$

Figure 5.1: The language L_1

5.1 Runtime Representations of Object-language Types

As we have seen in Section 4.3, values of interesting domains for which we encode properties and predicates (e.g., natural numbers in the previous section) are encoded as *types* in the meta-language. We will call such types in the meta-language *domain value types*. For example, the types `S, Z, IsNat, PlusRel` are such types (and type constructors). Domain value types are not formally different from any other meta-language types – the distinction of purely one of convention and use.

5.1.1 Types

The set of L_1 types is represented by a subset of Haskell types themselves. The semantic function $[\![\bullet]\!] : \tau \to$ **Set** gives the appropriate mapping from L_1 types to the types in the metalanguage, in this case Haskell. The type `Rep t` defines which Haskell types are permitted to be used as L_1 types: if, for some Haskell type `t`, we have a value of type `Rep t`, then this is a proof that there exists an L_1 type τ, such that $[\![\tau]\!] =$ `t`.

We will call this type *runtime type representations:* (or *reps*) a value of data-type `Rep t` represents type `t` (Figure 5.2).

```
1 data Rep t
2  = Rint (Equal t Int)
3  | Runit (Equal t ())
4  | ∀αβ. Rarr (Rep α) (Rep β) (Equal t (α→β))
5  | ∀αβ. Rpair(Rep α) (Rep β) (Equal t (α , β))
6
```

```
7  rint  :: Rep Int
8  rint   = Rint refl
9
10 runit :: Rep ()
11 runit = Runit refl
12
13 rarr  :: Rep a→Rep b→Rep (a→b)
14 rarr r1 r2 = Rarr r1 r2 refl
15
16 rpair :: Rep a→Rep b→Rep (a,b)
17 rpair r1 r2 = Rpair r1 r2 refl
```

The constructor `Rint :: Rep t` (line 2) contains the proof that the type `t` is equal to the type `Int`. Similarly, the constructor `Rarr` (line 4) contains representations of the domain and the codomain types α and β together with a proof that type `t` equals to $\alpha \rightarrow \beta$. The domain and codomain types are existentially quantified. The equality proof allows us to cast between `t` objects and the function space $\alpha \rightarrow \beta$ whenever we deconstruct the representation itself.

The important feature of *runtime type representations* (henceforth `Rep`s) is that they can be compared for equality. The function `testEq` compares to type `Rep`s (of types `t1` and `t2`) at runtime and if they are equal, constructs a proof of that equality. This proof can then be used to cast from values of type `t1` to values of type `t2`.

```
1  testEq :: Rep t1→Rep t2→Maybe (Equal t1 t2)
2  testEq (RUnit p1) (RUnit p2) = return  (p1 <> (sym p2))
3  testEq (RInt p1) (RInt p2) = return (p1 <> (sym p2))
4  testEq (RArr d1 c1 p1) (RArr d2 c2 p2) =
5       do { p3 <- testEq d1 d2
6          ; p4 <- testEq c1 c2
7          ; return (p1 <> (subTab p3 p4) <> (sym p2)) }
8  testEq (RPair d1 c1 p1) (RPair d2 c2 p2) =
9       do { p3 <- testEq d1 d2
10         ; p4 <- testEq c1 c2
11         ; return (p1 <> (subTab p3 p4) <> (sym p2)) }
12 testEq _ _ = Nothing
```

The base cases are quite simple. For example, the case comparing two representations of unit type

```
testEq (RUnit p1) (RUnit p2) = return  (p1 <> (sym p2))
```

The proof object `p1` has the type `Equal t1 ()`, and the proof object `p2` has the type `Equal t2 ()`. These proofs are easily combined to construct the proof `Equal t1 t2`: $\texttt{t1} \xRightarrow{\texttt{p1}} () \xRightarrow{\texttt{sym p2}} \texttt{t2}$.

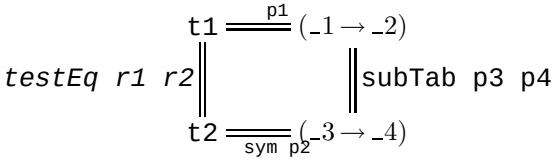
The other cases work by deconstructing the two `Rep`s in parallel, comparing their subparts for equality, and combining them into proofs of equality between the original `Rep`s. We examine the case for `RArr` (lines 4-7).

```
testEq (r1@(RArr d1 c1 p1)) (r2@(RArr d2 c2 p2)) =
     do { p3 <- testEq d1 d2
        ; p4 <- testEq c1 c2
        ; return (p1 <> (subTab p3 p4) <> (sym p2)) }
```

We start with proof objects `p1::Equal t1` $(\_1 \to \_2)$ and `p2::Equal t2` $(\_3 \to \_4)$. The first recursive call to `testEq` computes the proof object `p3::Equal` $\_1$ $\_3$, and the second recursive call computes the proof object `p4::Equal` $\_2$ $\_4$. The proofs `p2` and `p3` are combined by `subTab` to obtain the proof

```
subTab p3 p4 :: Equal (_1→_2) (_3→_4)
```

The final result is obtained by combining these proofs (using the proof combinators `trans` and `sym`), which we show graphically:



Type representations are a powerful programming tool. As we have seen before, domain value types encode interesting values in the system. Programming languages such as Haskell, however, do not allow computation to take place at the type level. Runtime comparison of type representations can be used to simulate this kind of computation. At runtime, a value of type `Rep t`, can be compared to some other value `Rep t'`. If they are equal, then we know that the domain value type `t` evaluates to `t'`, and can use the resulting equality proof to cast between the two. If the equality test fails, that means that the domain value type `t` would not evaluate to `t'` and the expression was not correctly typed in the first place, leaving the user the ability to gracefully exit the program. This technique has also been used to implement dynamic typing in a disciplined and type safe manner [4].

To demonstrate the use of the function `testEq`, consider the following small example:

```
1 increment :: Rep t→t→Maybe Int
2 increment rt i =
```

```
3   do { p <- testEq rt rint -- p:: Equal t Int
4      ; return ((a2b p1 i) + 1) }
```

The function `increment` expects two arguments: a representation of type `t`, and a value of type `t`. If the representation is an integer, `increment` increments the integer by one; otherwise, it returns `Nothing`.

This function relies on `testEq` (line 3) to compare the argument representation to `Rep Int`. If the comparison succeeds, the proof `p` can then be used to convert a `t` object into an integer and perform the addition. Otherwise, the monad simply propagates failure.

5.1.2 Expressions

Following the method demonstrated in Section 4.3, we can map the remaining syntactic definitions of L_1 into their corresponding Haskell data-types. First, we will define a number of types and type constructors that correspond to syntactic pre-terms of L_1.

```
1 newtype ABS t e   = ABS (Rep t) e
2 newtype VAR x     = VAR x
3 newtype APP e1 e2 = APP e1 e2
4 newtype LIT i     = LIT i
```

We call them "pre-terms" because up to this point there is no way to ensure that these types are combined in a syntactically correct way. For example, the following expression does not correspond to any valid L_1 term:

```
VAR (VAR (LIT String)) :: VAR (VAR (LIT String))
```

Now, we return from our digression and define an inductive judgment `IsExp::*` $\rightarrow$ `*`. This judgment defines what it means to be a well-formed syntactic expression. The intuition is that if we have a value of type `IsExp t`, then `t` is a domain value type representing some syntactic expression e at the type level. Furthermore, there is again a one-to-one correspondence between values of type `IsExp t` and the expression represented by `t`.

```
1 data IsExp t =
2      ∀n.     IsVar (IsNat n) (Equal t (VAR n))
3    | ∀e1 e2. IsApp (IsExp e1) (IsExp e2) (Equal t (APP e1 e2))
4    | ∀tdom e.IsAbs (Rep tdom) (IsExp e) (Equal t (ABS tdom e))
5    | ∀n.     IsLit n (Equal t (LIT n))
```

```
6
7  isVar :: IsNat n -> IsExp (Var n)
8  isVar n = V n refl
9
10 isApp :: IsExp e1 -> IsExp e2 -> IsExp (APP e1 e2)
11 isApp e1 e2 = IsApp e1 e2 refl
12
13 isAbs :: Rep t -> IsExp e -> IsExp (ABS t e)
14 isAbs t e   = IsAbs t e refl
15
16 isLit :: a -> IsExp (LIT a)
17 isLit n = IsLit n refl
```

The type constructor `IsExp` plays the same role for expression, as the type constructor `IsNat` for the naturals. For example,

```
1  exp1 :: IsExp (ABS Int (ABS (Int→Int) (APP (VAR Z) (VAR (S Z)))))
2  exp1 = isAbs rint
3               (isAbs (rarr rint rint)
4                      (isApp (isVar z) (isVar (s z))))
```

Well-formed type assignments can also be represented at the level of types.

```
1  data IsGamma gamma =
2             IsEmpty (Equal gamma ())
3    |  ∀g t. IsGammaExt (IsGamma gamma) (Rep t) (Equal gamma (g,t))
4
5
6  isEmpty :: IsGamma ()
7  isEmpty  = IsEmpty refl
8
9  isGammaExt :: IsGamma g→Rep t→IsGamma (g,t)
10 isGammaExp g r = IsGammaExp g r refl
```

The purpose of this section has been to demonstrate that more complex domain value types (e.g., those representing expressions, type assignments etc.) can be represented and manipulated in the paradigm we propose. In what follows, we will not use this particular encoding as it is not needed – it has been presented here just for completeness' sake.

5.2 Judgments: representing well-typed terms

We begin with a few preliminary observations. First, we recall that there is a set of derivations of the judgment $\Gamma \vdash e : \tau$. This set is defined inductively by the rules in Figure 5.1. Now, we examine the correspondences between definitions of various sets (Figure 5.1) and the Haskell implementation.

The set of types τ, is encoded by Haskell *types* themselves. For example, the L_1 type $\mathsf{Int} \to \mathsf{Int}$ is represented by the Haskell type `Int → Int`. Thus, the semantic function in Figure 5.1, ($[\![\tau]\!]$) is then simply the identity function, since the meanings of terms of a certain type will be mapped into exactly the same type again.[2] Similarly, type assignments (contexts, Γ) are represented in Haskell using the Haskell product type. For example the type assignment $\langle\rangle, \mathsf{Int}, \mathsf{Int} \to \mathsf{Int}$ is represented by the Haskell nested product `(((),Int),Int → Int)`. The underlying semantics of these types, in turn, is provided by the semantics of the underlying language, namely Haskell. Now, we are ready to present the actual encoding of type judgments and their proofs.

The judgment $\Gamma \vdash e : \tau$ is implemented by a Haskell type constructor `Exp g t` of kind `* → * → *`. Each derivation rule from the top of Figure 5.1 is represented by a constructor of the `Exp` data-type. We can read the type `e :: Exp g t` as "Under the type assignment `g`, there is an expression `e` that has type `t`." Figure 5.2 summarizes the relevant definitions for the Haskell encoding.

We have shown how the syntactic expressions of L_1 can be encoded at type level as a judgment `IsExp` (Section 5.1.2). Following the pattern described in the natural numbers example (Section 4.3), one might expect that judgments would be encoded by a *ternary* type constructor of kind `* → * → * → *`, so that $\Gamma \vdash e : \tau$ corresponds to `Exp g e t`.

Instead, in our encoding, we will opt for an encoding of the L_1 typing judgment that does not require the L_1 expressions to appear in its type. This is because the the expression part of the judgment is uniquely determined by the type assignment, the type of the expression, and the structure of the typing derivation (See Proposition 3).[3]

How is the set of typing judgments encoded in Haskell? Each constructor of `Exp` corresponds for a derivation rule of the static semantics of L_1. We examine each data-constructor of `Exp` in detail below.

Variables. If we examine the judgments of Figure 5.1 for variable cases, we will notice that the two cases for variables are defined inductively on the natural number that represents the distance of the variable from its binding site. To simulate this induction on the bound variable indices, rather than on the structure of

[2] This is due to the fact that types in Haskell and types in L_1 are very similar. For some other language whose types differ from Haskell's, one must find a less trivial mapping into Haskell types.

[3] Another way of saying this is that the type judgment is syntax directed.

```
1  data Exp e t
2    = Lit Int (Equal t Int)
3    | V (Var e t)
4    | ∀αβ. Abs (Rep α) (Exp (e,α) β) (Equal t (α→β))
5    | ∀α.  App (Exp e (α→t)) (Exp e α)
6    | ∀αβ. Pair (Exp e α) (Exp e β) (Equal t (α,β))
7    | ∀αβ. Pi1 (Exp e (α,β)) (Equal t α)
8    | ∀αβ. Pi2 (Exp e (α,β)) (Equal t β)
9
10 data Var e t
11  = ∀γ.  Z (Equal e (γ,t))
12  | ∀γα. S (Var γ t) (Equal e (γ,α))
13
14 data Rep t
15  = Rint (Equal t Int)
16  | Runit (Equal t ())
17  | ∀αβ. Rarr (Rep α) (Rep β) (Equal t (α→β))
18  | ∀αβ. Rpair (Rep α) (Rep β) (Equal t (α , β))
```

Figure 5.2: Haskell implementation of `Exp`.

expressions, we define an auxiliary data-type `Var` of kind `* → * → *`.

```
data Var e t
 = ∀γ.  Z (Equal e (γ,t))
 | ∀γα. S (Var γ t) (Equal e (γ,α))
```

We show the derivation rule and the definition of the constructor side by side:

$\dfrac{}{\gamma, \mathtt{t} \vdash 0 : \mathtt{t}}$	`| ∀γ.  Z (Equal e (γ,t))`
$\dfrac{\gamma \vdash n : \mathtt{t}}{\gamma, \alpha \vdash (n+1) : \mathtt{t}}$	`| ∀γα. S (Exp γ t)` `(Equal e (γ,α))`

Just as in the judgment of Figure 5.1, there are two cases:

1. First, there is the constructor `Z`. This constructor translates the inductive definition directly: as its argument it takes a proof that there exists some environment γ such that the environment `t` is equal to γ extended by `t`.

2. The second constructor, `S` takes a proof that `(Var γ t)`, and as its second argument it takes the proof that the environment `e` is equal to the pair (γ, α), where both γ and α are existentially quantified.

The names `S` and `Z` are chosen to show how the proofs of the variable judgment are structurally the same as the natural number indices. Finally, the sub-proofs for the variable case are "plugged" into the definition of `Exp e t` using the constructor `V`.

Finally, for the `Var` data-type we define the two smart constructors:

```
z :: Var (a,b) b
z = Z refl

s :: (Var e t)→(Var (e,a) t)
s v = S v refl
```

Abstraction. The typing rule for abstraction is $\frac{\Gamma, \alpha \vdash e : \beta}{\Gamma \vdash \lambda\alpha.e : \alpha \to \beta}$. Translation of this derivation into Haskell is as follows:

```
data Exp e t = . . .
   (∀α β. Abs (Rep α) (Exp (e,α) β) (Equal t (α→β)))
```

Intuitively, we can create a typing derivation using the `Abs` rule if there exist some Haskell types α and β such that

a. We can provide a representation of the type α. This part directly corresponds to the requirement that the syntax of the lambda expression carry the type of the argument variable.

b. We can provide a proof of the judgment `Exp (e,`α`)` β. This is equivalent to the proof of the antecedent $\Gamma, \alpha \vdash e : \beta$: the abstraction is well-typed if the body of the abstraction is well-typed in an environment extended with the domain type, (e, α) and has the codomain type β.

c. And finally, if we can construct the proof that the argument type `t` is equal to the type $\alpha \to \beta$. Haskell's system of data-types forces each data constructor function to return a `Exp e t`. This equality proof argument allows us to work around this restriction, since the proof that `t` equals $\alpha \to \beta$ allows us to cast a `t` into the type $\alpha \to \beta$.

The smart constructor for abstraction is defined as follows:

```
abs :: Rep t1→Exp (e,t1) t2→Exp e (t1→t2)
abs typ body = Abs typ body refl
```

Application. The definition of the data constructor for application is given below.

```
  | ∀α. App (Exp e (α → t)) (Exp e α)
```

It takes two arguments: first is the proof of judgment of the function expression – this expression has an arrow type $\alpha \rightarrow$ `t`; the second argument is the proof of the judgment for the argument to which the function is applied. It's type must be identical to the type α of the function domain.

Since this constructor does not contain any equality proofs, there is no need for a smart constructor. For syntactic uniformity with other smart constructors, a trivial smart constructor is defined for this case:

```
app :: Exp e (t1 → t2) → Exp e t1 → Exp e t2
app = App
```

Examples. We show a couple of examples of L_1 typing judgments in Haskell and their proofs. First thing to note is that the proofs are constructed using the lower-case smart constructors; the use of these functions forces the Haskell type system to automatically infer the correct shape of the arguments to the type constructor `Exp` whose value is being constructed. First, we define the value `example1`.

```
example1 :: Exp e (Int → (Int → Int) → Int)
example1 = -- λx.λf. f x
  abs rint
      (abs (rarr rint rint)
           (app (V z) (V (s z))))
```

The definition `example1` corresponds to the following L_1 type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Diamond, \mathsf{Int} \vdash 0 : \mathsf{Int}}(\mathsf{VarZ})}{\Diamond, \mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int} \vdash 1 : \mathsf{Int}}(\mathsf{VarS})}{\Diamond, \mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int} \vdash \mathsf{Var}\ 1 : \mathsf{Int}}(\mathsf{Var}) \quad \dfrac{\dfrac{}{\Diamond, \mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int} \vdash 0 : (\mathsf{Int} \rightarrow \mathsf{Int})}(\mathsf{VarZ})}{\Diamond, \mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int} \vdash \mathsf{Var}\ 0 : (\mathsf{Int} \rightarrow \mathsf{Int})}(\mathsf{Var})}{\Diamond, \mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int} \vdash (\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1) : \mathsf{Int}}(\mathsf{App})$$

$$\dfrac{\Diamond, \mathsf{Int}, \mathsf{Int} \rightarrow \mathsf{Int} \vdash (\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1) : \mathsf{Int}}{\Diamond, \mathsf{Int} \vdash \lambda \mathsf{Int} \rightarrow \mathsf{Int}.\ (\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1) : ((\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int})}(\mathsf{Abs})$$

$$\dfrac{\Diamond, \mathsf{Int} \vdash \lambda \mathsf{Int} \rightarrow \mathsf{Int}.\ (\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1) : ((\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int})}{\Diamond \vdash \lambda \mathsf{Int}.\ \lambda \mathsf{Int} \rightarrow \mathsf{Int}.\ (\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1) : (\mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int})}(\mathsf{Abs})$$

Also, note that the following definition, `example2`, shows how to write proofs of `Exp` judgments for open terms.

```
example2 :: Exp ((a,b),(Int,b) → c) (Int → c)
example2 = -- λx2. f1 (x0, x2)
  abs rint
     (app  (V (s z)) (pair (V z) (V (s (s z)))))
```

Variable indices 1 and 2 are used in the body of the abstraction. This forces the type of the environment argument to `Exp` to grow to accommodate a correct type assignment for the free variables. The definition `example2` corresponds to the following L_1 typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots((\mathsf{Int}\times b)\to c)\vdash 0:((\mathsf{Int}\times b)\to c)}(\mathrm{VarZ})}{\cdots((\mathsf{Int}\times b)\to c),\mathsf{Int}\vdash 1:((\mathsf{Int}\times b)\to c)}(\mathrm{VarS})}{\Diamond,b,((\mathsf{Int}\times b)\to c),\mathsf{Int}\vdash \mathsf{Var}\ 1:((\mathsf{Int}\times b)\to c)}(\mathrm{Var}) \quad \dfrac{\dfrac{\vdots}{\cdots\vdash \mathsf{Var}\ 0:\mathsf{Int}}(\mathrm{Var}) \quad \dfrac{\vdots}{\cdots\vdash \mathsf{Var}\ 2:b}(\mathrm{Var})}{\cdots\vdash(\mathsf{Var}\ 0,\mathsf{Var}\ 2):\mathsf{Int}\times c}(\mathrm{Pair})}{\Diamond,b,((\mathsf{Int}\times b)\to c),\mathsf{Int}\vdash(\mathsf{Var}\ 1)\ (\mathsf{Var}\ 0,\mathsf{Var}\ 2):(\mathsf{Int}\to c)}(\mathrm{App})}{\Diamond,b,((\mathsf{Int}\times b)\to c)\vdash\lambda\mathsf{Int}.\ (\mathsf{Var}\ 1)\ (\mathsf{Var}\ 0,\mathsf{Var}\ 2):(\mathsf{Int}\to c)}(\mathrm{Abs})$$

One should note that type variables that occur in the type of `example2` are *not* part of the type system of L_1; rather, they are meta-variables. Intuitively, this corresponds to a whole *family* of L_1 judgments, where arbitrary L_1 types can be substituted for meta-variables b and c.

5.2.1 Interpreter

The interpreter function is, in a way, the simplest of all the artifacts of the language implementation in this style. It is a function of type `Exp e t → e → t`, whose definition is shown in Figure 5.3.

In this function the equality proofs that proofs of judgments contain become essential.

```
(2) eval (Lit i p) env = b2a p i
       -- p :: Equal t Int
       -- i :: Int
       -- b2a p i :: t
```

In line 2, the `eval` function must return a result of type `t`, but all we have is the integer `i`. However, we also have the proof `p :: Equal t Int`. Now we can use the function `b2a` to obtain `(b2a p i)` which has the type `t`.

The the variable case (line 3) simply passes control to the auxiliary function `evalVar`.

```
eval (V v) env = evalVar v env

evalVar :: (Var e t) → e → t
evalVar (Z p) env = snd (a2b p env)     -- p :: Equal e (_1,t)
evalVar (S v p) env = evalVar v (fst (a2b p env))
   -- env       :: e
   -- v         :: Var _1 t
```

```
  -- p         :: Equal e (_1,_2)
  -- a2b p env :: (_1,_2)
```

This function performs the appropriate projection from the environment: in line 11, we first use the proof object `p :: Equal e (`γ`,t)` to cast the environment `e` into type $(\gamma, \texttt{t})$, and then to project the second element of type `t`. Line 12 implements the weakening case for variables. Again, the equality proof `p` is used to cast the environment to a pair, and pass the sub-environment to the recursive call to `evalVar`.

The case for application is defined as follows (in line 4):

```
eval (App f x) env = (eval f env) (eval x env)
  -- f :: Exp e (_1→t)
  -- x :: Exp e _1
  -- eval f env :: _1→t
  -- eval x env :: _1
  -- (eval f env) (eval x env) :: t
```

First, the function part of the application is evaluated, obtaining a function value of type $\alpha \rightarrow \texttt{t}$; next, the argument is evaluated obtaining a value of type α. Finally the resulting function is applied, obtaining a result of type `t`. It is worth noting that in this case the function `eval` is called recursively at two different instances, namely `Exp e (`$\alpha \rightarrow$ `t)` and `Exp e` α, requiring the use of *polymorphic recursion*.

```
1  eval :: (Exp e t) → e → t
2  eval (Lit i p) env = b2a p i
3  eval (V v) env = evalVar v env
4  eval (App f x) env = (eval f env) (eval x env)
5  eval (Abs r body p) env = b2a p (\ x → eval body (env,x))
6  eval (Pair x y p) env =  b2a p (eval x env, eval y env)
7  eval (Pi1 e p) env =   b2a p (fst (eval e env))
8  eval (Pi2 e p) env =   b2a p (snd (eval e env))
9
10 evalVar :: (Var e t) → e → t
11 evalVar (Z p) env = snd (a2b p env)
12 evalVar (S v p) env = evalVar v (fst (a2b p env))
```

Figure 5.3: L_1: the interpreter `eval`

Other cases of `eval` are similar to the ones already discussed above, and will not be elaborated in detail. The general pattern could be summarized as follows. The function eval takes apart a proof of a judgment (`Exp` or `Var`) to produce a value: the type of the value produced is contained in the type index of the judgment. The proof of the judgment must contain sufficient equality proofs that can be used to circumvent

typing problems that arise by casting. The inductive nature of the judgment proofs often requires that `eval` be called recursively at different types, so the use of polymorphic recursion is essential.

5.2.2 Type-checker

```
1  type Name = String
2
3  data E
4    = I Int    | A E E  | Lam Name T E
5    | Var Name | P E E  | P1 E
6    | P2 E
7  data T = ∀α. T (Rep α)
8  tint,tunit :: T
9  tint = T rint
10 tunit = T runit
11 tarr       :: T → T → T
12 tarr x y =
13   case (x,y) of (T a,T b) → T (rarr a b)
14 tpair x y  :: T → T → T
```

Figure 5.4: Syntactic pre-expressions and types

In Section 5.2 we have shown how the data-type `Exp e t` encodes only well-typed L_1 terms.[4] In Section 5.2.1 we have presented an interpreter which maps well-typed L_1 terms of type `Exp e t` into corresponding values of type `t`. One part that is missing in this language implementation is some kind of *parsing* or *type-checking* function. Such a function must take as its arguments either strings, or simple pre-expressions of L_1, and produce `Exp e t` values if the input terms are well-typed (or if they are textual representations of well-typed L_1 terms).

We make a small digression here to make an observation about the syntactic pre-terms `E` (Figure 5.4). For increased human readability the pre-expressions do not use de Bruijn style of variable representation. Thus the type-checking function converts these terms with variable names to the nameless notation of `Exp` judgment proofs. This is easily accomplished by simply keeping a history of binding occurrences of variables as we descend down the term, then computing its position in the list at variable use sites. Second, pre-expressions `E` carry type annotations on bound variables in λ-abstractions. For this we need some syntactic representation of types. We resort to a very useful and concise trick: a type of syntactic representations of L_1 types will simply be the data-type `T`, where $T = \exists\alpha.\texttt{Rep}\,\alpha$. This way, converting the

[4] To be precise, values of this data-type encode proof derivations of the typing judgments of L_1, but since for each well-typed L_1 expression in a given context there is only one such derivation, we can treat the proofs as standing for their corresponding terms. We will use the term "well-typed expression" for such a proof where the correspondence is clear from the context.

syntactic types `T` into `Rep`s is accomplished by simply "unpacking" existential package type `T`.

Having defined syntactic pre-terms, we encounter a problem, however, when we try to give a type to the type-checking function:

```
tc :: E→Exp ?1 ?2
```

The problem is that types to be used in place of $?_1$ and $?_2$ are different depending on the values of the `E` argument, which means that the function `tc` could not return a single type, but rather a whole family of types. For example, for an input term $\lambda x : \mathsf{Int}.\, x$ it must return `Exp e (Int→Int)`, while for the input term 4 it must return `Exp e Int`.

Fortunately, using existential types we can indeed give a type to the function `tc` used above. This type is:

```
tc :: ··· → Maybe (∃αβ. ((Rep β), (Rep α),
                         (Exp α β)))
```

One thing to note is that in Haskell we must encode existential types as data-types. This is why we define the data-type `J f`, which takes a binary type constructor `f`, and encodes $\exists\alpha\beta.\,(\mathsf{Rep}\ \beta)(\mathsf{Rep}\ \alpha)(\mathtt{f}\ \alpha\ \beta)$. Then, `f` can be instantiated either with `Exp` to obtain the range type of `tc`, or with `Var` to obtain the range type of `lookup`(shown later). The full implementation of the function `tc` is given in Figure 5.5.

```
data J f = ∀αβ. J (Rep β) (Rep α) (f α β)

tc :: [Name] → E → T → Maybe (J Exp)
```

The first argument to the function is a list of variable names, which is used to compute the appropriate variable indices. The second argument is, of course, the pre-expression for which a judgment will be constructed. The third argument of `tc` is the initial type assignment giving types for free variables in the input expression. Conceptually, this is a list of types corresponding to the types whose indices are listed in the first argument. However, we will use a single nested pair type to encode this list in order to make our definitions more compact.

The `Maybe` type of the codomain represents the possibility that the input may not be well-typed and therefore no `Exp` can be produced. In addition to an `(Exp α β)` it is necessary that the function return a runtime representation of the types of the environment and the result as well, so they too are included in the type of `J` above.

The type `J Exp` (line 1) is defined as a representation of `∃et.(Rep t) (Rep e) (Exp e t)` and

`J Var` for ∃`et.(Rep t) (Rep e) (Var e t)`, since Haskell allows the use existential types only in data-type definitions.

Now let us examine some of the cases for which the function `tc` is defined. The case for literals (line 4) is quite simple: the type environment argument is unpacked and stored as the type representation of the environment.

```
(4) tc vs (I i) (T env) = return (J rint env (lit i))
```

Type representation `rint` is used to encode the type of the expression itself. The proof of the typing judgment itself is (`lit i`). These three values are packed up together and returned as a result of type `J Exp`.

Next case is the abstraction (lines 8-10).

```
(8)  tc vs (Lam name t e) gamma =
       do { J rcod (Rpair renv rdom p1) j ← tc (name:vs) e (tpair gamma t)
(10)     ; return (J (rarr rdom rcod) renv (lam rdom (castTa_ p1 j))) }
```

Here we first recursively construct proof for the typing judgment of the body of the λ-abstraction in the type assignment extended by the domain type. Then, another package is constructed as a proof of the judgment for the abstraction. In line 10 the combinator `castTa_` is used to cast `j`, which has the type `Exp _e cod` to `Exp (env,dom) cod`, where `rdom :: Rep dom`, `rcod ::Rep cod` and `renv :: Rep env`. Such use of casting and other equality combinators is necessary to ensure that existential variables do not escape the scope of their unpacking.

The case for application (lines 11-17) is more complex.

```
(11) tc vs (A f a) gamma =
       do { J rf env1 f ← tc vs f gamma  -- rf :: Rep f
          ; J ry env2 y ← tc vs a gamma  -- ry :: Rep y
          ; Rarr a b p1 ← return rf
          ; p2 ← testEq ry a             -- p2 :: Equal y a
          ; p3 ← testEq env2 env1
(17)      ; return (J b env1 (app (castTa p1 f) (castTab p3 p2 y))) }
```

It uses `testEq` in a number of places to ensure that the representation of the types returned by the recursive calls match. For example, the type of the domain of the function must be equal to the representation of

the type of the argument. Then, various casting operators use the proofs of equality returned by those tests to correctly type the resulting judgment. The function `testEq` ensures that if any of these equalities fail, the entire type-checking function fails as well.

Finally, we show the variable case.

```
(5) tc vs (Var str) gamma =
(6)   do { J t e j ← lookup str vs gamma
(7)      ; return (J t e (V j)) }

(24) lookup :: [Char] → [[Char]] → T → Maybe (J Var)
     lookup nm [] env = Nothing
     lookup nm (n:ns) (T (Rpair a b p1)) =
        if eqStr nm n
           then return (J b (rpair a b) z)
           else do { J ty rgamma j ← lookup nm ns (T a)
                   ; return(J ty (rpair rgamma b) (s j))}
(31) lookup nm ns env = Nothing
```

As with `eval`, the variable case (lines 5-7) is implemented using an auxiliary function to handle the induction on variable indices: the function `tc` passes control to the function `lookup` (lines 24-31). The function `lookup` constructs the sub-derivation of type `J Var`, by searching down the list of variable names and building appropriate `Var` index. Once `lookup` returns, its result is unpacked (line 6) and repackaged as a `J Exp`.

5.3 Pattern matching and L_1^+

In this section, we shall extend the language L_1 with sum types and pattern matching. We shall call the language so obtained L_1^+. The motivation for this step is twofold:

1. Patterns are an interesting feature of most modern functional programming language. Demonstrating that patterns can be easily and elegantly integrated into our implementation framework is a further demonstration of its usefulness and power.

2. Pattern matching introduces the notion of *failure* into the semantics of the language. Such failure is one of the simplest *computational effects* that can be introduced into a programming language. Concentrating on such a simple effects will help motivate our further forays into this area.

```
1  data J f = ∀αβ. J (Rep β) (Rep α) (f α β)
2
3  tc :: [Name] → E → T → Maybe (J Exp)
4  tc vs (I i) (T env) = return (J rint env (lit i))
5  tc vs (Var str) gamma =
6    do { J t e j ← lookup str vs gamma
7       ; return (J t e (V j)) }
8  tc vs (Lam name t e) gamma =
9    do { J rcod (Rpair renv rdom p1) j ← tc (name:vs) e (tpair gamma t)
10      ; return (J (rarr rdom rcod) renv (lam rdom (castTa_ p1 j))) }
11 tc vs (A f a) gamma =
12   do { J rf env1 f ← tc vs f gamma  -- rf :: Rep f
13      ; J ry env2 y ← tc vs a gamma  -- ry :: Rep y
14      ; Rarr a b p1 ← return rf
15      ; p2 ← testEq ry a             -- p2 :: Equal y a
16      ; p3 ← testEq env2 env1
17      ; return (J b env1 (app (castTa p1 f) (castTab p3 p2 y))) }
18 tc vs (P x y) gamma =
19   do { J rx env1 xexp ← tc vs x gamma  -- rf :: Rep f
20      ; J ry env2 yexp ← tc vs y gamma  -- ry :: Rep y
21      ; p1 ← testEq env2 env1
22      ; return(J (rpair rx ry) env1 (pair xexp (castTab p1 refl yexp)))}
23
24 lookup :: [Char] → [[Char]] → T → Maybe (J Var)
25 lookup nm [] env = Nothing
26 lookup nm (n:ns) (T (Rpair a b p1)) =
27    if eqStr nm n
28       then return (J b (rpair a b) z)
29       else do { J ty rgamma j ← lookup nm ns (T a)
30               ; return(J ty (rpair rgamma b) (s j))}
31 lookup nm ns env = Nothing
```

Figure 5.5: Typechecking function for L_1

5.3.1 Syntax of L_1^+

First, the definition of L_1^+ types is obtained by extending L_1 types with sums:

$$\tau \in \mathbb{T} ::= \cdots \mid \tau_1 + \tau_2$$

Patterns. We shall first define a notion of *pattern* that will allow us a more succinct and flexible notation for eliminations of both sums and products, modeled after similar constructs in functional languages such as Standard ML or Haskell.

The set of patterns is defined as follows:

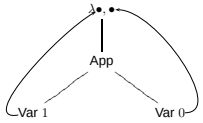
$$p \in \mathbb{P} ::= \bullet_\tau \mid \mathsf{Inl}\ p \mid \mathsf{Inr}\ p \mid (p_1, p_2)$$

Patterns can either be variable bindings ($\bullet_\tau$) which are annotated by the type of the values they bind, the left or right case of the sum constructor, or pairs of patterns. In the text that follows, we shall omit the explicit type annotations whenever they are discernible from the context.

$$\frac{}{\Gamma \vdash \bullet_\tau : \tau \Rightarrow \Gamma, \tau}(\text{Var}) \qquad \frac{\Gamma \vdash p : \tau_1 \Rightarrow \Gamma_2}{\Gamma \vdash \mathsf{Inl}\ p : \tau_1 + \tau_2 \Rightarrow \Gamma_2}(\text{Inl})$$

$$\frac{\Gamma \vdash p : \tau_2 \Rightarrow \Gamma_2}{\Gamma \vdash \mathsf{Inr}\ p : \tau_1 + \tau_2 \Rightarrow \Gamma_2}(\text{Inr}) \qquad \frac{\Gamma \vdash p_1 : \tau_1 \Rightarrow \Gamma_2\ \Gamma_2 \vdash p_2 : \tau_2 \Rightarrow \Gamma_3}{\Gamma \vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow \Gamma_3}(\text{Pair})$$

The intuition behind the pattern checking relation $\Gamma_1 \vdash p : \tau \Rightarrow \Gamma_2$ is: "under the type assignment Γ_1, the pattern p deconstructs an expression of type τ yielding an extended type assignment Γ_2." The positional style for naming variables that we have adopted throughout this chapter means that variables bound in patterns do not have names. Since more than one variable can be bound in pattern, we must make a decision as to what numerical indices those variables will be referred to: we chose that the "furthest" binding site is the leftmost-bottommost variable.[5] The picture in Figure 5.6 illustrates the binding structure of the term $\lambda(\bullet, \bullet).\,(\mathsf{Var}\ 1, \mathsf{Var}\ 0)$, where the curved lines point to the binding site of variables in the body of an abstraction.



$$\frac{\dfrac{\Gamma \vdash \bullet_{\mathsf{Int}} : \mathsf{Int} \Rightarrow \Gamma, \mathsf{Int} \quad \Gamma, \mathsf{Int} \vdash \bullet_{\mathsf{Int}\to\mathsf{Int}} : \mathsf{Int} \Rightarrow \Gamma, \mathsf{Int}, \mathsf{Int} \to \mathsf{Int}}{\Gamma \vdash (\bullet_{\mathsf{Int}}, \bullet_{\mathsf{Int}\to\mathsf{Int}}) : (\mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int})) \Rightarrow \Gamma, \mathsf{Int}, (\mathsf{Int} \to \mathsf{Int})}}{\Gamma \vdash \mathsf{Inl}\ (\bullet_{\mathsf{Int}}, \bullet_{\mathsf{Int}\to\mathsf{Int}}) : (\mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int})) + \mathsf{Int} \Rightarrow \Gamma, \mathsf{Int}, (\mathsf{Int} \to \mathsf{Int})}$$

Figure 5.6: Binding multiple variables in patterns.

The definition of the pattern checking relation (case for pairs) reflects this – left sub-pattern bindings precede right sub-pattern bindings in the augmented type assignments. For example, the Figure 5.6 gives the derivation rules for proofs of the pattern judgment for $\mathsf{Inl}\ (\bullet_{\mathsf{Int}}, \bullet_{\mathsf{Int}\to\mathsf{Int}})$.

The next step is to extend the λ-abstractions of L_1 to work with patterns. Note that the patterns in λ-abstractions do not admit alternatives, and we will delay the discussion of the semantics of pattern matching failure until later section when we discuss the case expressions. The syntactic form for the new λ-abstraction is as follows:

$$e \in \mathbb{E} ::= \cdots \mid \lambda p.e$$

The typing rule incorporates the new pattern typing judgments:

$$\frac{\Gamma \vdash p : \tau_1 \Rightarrow \Gamma_2 \quad \Gamma_2 \vdash e : \tau_2}{\Gamma \vdash \lambda p.e : \tau_1 \to \tau_2}(\text{Abs})$$

[5] Alternatively, we could say that the rightmost-uppermost variable is the one whose index is 0.

The new-style abstractions include the old-style abstraction virtually unchanged: $\Gamma \vdash \lambda\bullet_{\mathsf{Int}.\mathsf{Var}\ 0} : (\mathsf{Int} \to \mathsf{Int})$. However, now we can have more complicated abstractions: $\Gamma \vdash \lambda(\bullet_{\mathsf{Int}}, \bullet_{\mathsf{Bool}}).(\mathsf{Var}\ 0, \mathsf{Var}\ 1) : (\mathsf{Int} \times \mathsf{Bool} \to \mathsf{Bool} \times \mathsf{Int})$.

Sums. There are two new expression forms that are used as introduction constructs for sums, the two injections Inl and Inr. A case construct is used for sum elimination:

$$e \in \mathbb{E} ::= \cdots \mid \mathsf{Inl}_{\tau_1\ \tau_1}\ e \mid \mathsf{Inr}_{\tau_1\ \tau_2}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \overline{p_n \to e_n}$$

One thing to note is that the case expression takes an arbitrary number of matches of patterns that are the same as the ones introduced for λ-abstractions: they can be incomplete and/or nested to an arbitrary depth.

The typing rules for sum introduction and elimination are given below:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{Inl}_{\tau_1\ \tau_2}\ e : (\tau_1 + \tau_2)}(\text{Inl}) \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{Inr}_{\tau_1\ \tau_2}\ e : (\tau_1 + \tau_2)}(\text{Inr})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \overline{\Gamma \vdash p_n : \tau_1 \Rightarrow \Gamma_n \quad \Gamma_n \vdash e_n : \tau_2}}{\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \overline{p_n \to e_m} : \tau_2}(\text{Case})$$

5.3.2 Semantics of L_1 with Patterns

The semantics of sum types is easy to give in the categorical style we have used in Section 5.0.1. The meaning of a sum type is the disjoint sum of the meanings of the two summands:

$$\begin{aligned}
[\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] + [\![\tau_2]\!] \\
[\![\Gamma \vdash \mathsf{Inl}\ e : \tau_1 + \tau_2]\!] &= \mathbf{Inl}\ [\![\Gamma \vdash e : \tau_1]\!] \\
[\![\Gamma \vdash \mathsf{Inr}\ e : \tau_1 + \tau_2]\!] &= \mathbf{Inr}\ [\![\Gamma \vdash e : \tau_2]\!]
\end{aligned}$$

The addition of pattern matching to L_1 introduces a notion of *pattern matching failure* to the semantics of L_1 programs. The notion of failure may manifest itself in two (related) ways:

1. *Global failure.* Pattern matching may fail when no pattern can be found to deconstruct a particular value. This may occur, for example, in λ-expressions (or incomplete case expressions), such as $(\lambda(\mathsf{Inl}\ \bullet).\ \mathsf{Var}\ 0)\ (\mathsf{Inr}\ 10)$. In case of such a failure, the meaning of the program is undefined.

2. *Local failure.* Pattern matching may fail as one of a number of alternatives in a case expression. Local failure may, or may not, be promoted into a global failure: if one of a number pattern matches in a case expression fails, the control should be passed to other matches, until one of them succeeds. If there no such succeeding patterns, a global failure should take place.

The problem posed by introducing effects such as failure into the semantics of programming languages is that the entire semantic definition must be "overhauled," in order to properly propagate the effect of failure throughout the meaning of the program. We note here that local failure is more benign in this sense then global failure – it is possible to statically ensure that all pattern matches are complete, so that local failure is never promoted into a global failure. Encoding such a static restriction in the type system is an interesting problem for future work.

One way to structure denotational semantic definitions so as to be able to manage effects in a more clean, generic and modular way is to use *monads* ([84, 83, 72]).

The denotations of L_1 programs augmented by pattern matching are the meanings of the L_1 types augmented by a special value Fail indicating failure of pattern matching somewhere in the program. Thus, definition of the monad $\mathcal{M}$ for our purposes would be

$$\begin{aligned}
\mathcal{M}\ A &= A + \{\mathsf{Fail}\} \\
\mathsf{return}_{\mathcal{M}}\ e &= \mathbf{Inl}\ e \\
(\mathbf{Inr}\ \mathsf{Fail})\ \star_{\mathcal{M}}\ f &= \mathbf{Inr} \\
(\mathbf{Inl}\ v)\ \star_{\mathcal{M}}\ f &= f\ v
\end{aligned}$$

Two non-proper morphisms, `fail` and []($\llbracket$pronounced "fatbar"), are also defined.

$$\begin{aligned}
\mathsf{fail}_{\mathcal{M}} &= \mathbf{Inr}\ \mathsf{Fail} \\
(\mathbf{Inr}\ \mathsf{Fail}) \ \llbracket\ m &= m \\
m_1 \ \llbracket\ m_2 &= m_1
\end{aligned}$$

The first, fail, represents a failing computation. The second, $\llbracket$, is a biased (left) choice operator (also called "fatbar"): given two computations, its value is the first one, unless it fails, in which case it returns the second computation. (For a detailed discussion of semantics of patterns in Haskell, and of "fatbar", see [56]).

Finally, we can define the meaning of types of L_1^+ in a new monad-based framework.

$$\begin{aligned}
[\![\mathsf{Int}]\!] &= \mathbb{N} \\
[\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] + [\![\tau_2]\!] \\
[\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to \mathcal{M}\ [\![\tau_2]\!]
\end{aligned}$$

Sums in L_1^+ are mapped to the (set theoretic) sums in the meta-language. One notable difference from the semantics of L_1 types is that the function space has been changed so that its domain is $\mathcal{M}\ [\![\tau_2]\!]$. This reflects the fact that functions suspend computations and their effects: after a function is applied, computing the result of the range type $[\![\tau_2]\!]$ may also result in effects that the monad $\mathcal{M}$ hides.

The semantics of patterns is defined as follows:

$$
\begin{array}{ll}
[\![\Gamma \vdash p : \tau \Rightarrow \Gamma']\!] & : [\![\tau]\!] \to [\![\Gamma]\!] \to \mathcal{M}\ [\![\Gamma']\!] \\
[\![\Gamma \vdash \bullet_\tau : \tau \Rightarrow \Gamma, \tau]\!]\ v\ \rho & = \mathsf{return}_{\mathcal{M}}\ (\rho, v) \\
[\![\Gamma \vdash \mathsf{Inl}\ p : (\tau_1 + \tau_2) \Rightarrow \Gamma']\!]\ (\mathbf{Inl}\ v)\ \rho & = [\![\Gamma \vdash p : \tau_1 \Rightarrow \Gamma']\!]\ v\ \rho \\
[\![\Gamma \vdash \mathsf{Inl}\ p : (\tau_1 + \tau_2) \Rightarrow \Gamma']\!]\ (\mathbf{Inr}\ v)\ \rho & = \mathsf{fail}_{\mathcal{M}} \\
[\![\Gamma \vdash \mathsf{Inr}\ p : (\tau_1 + \tau_2) \Rightarrow \Gamma']\!]\ (\mathbf{Inr}\ v)\ \rho & = [\![\Gamma \vdash p : \tau_2 \Rightarrow \Gamma']\!]\ v\ \rho \\
[\![\Gamma \vdash \mathsf{Inr}\ p : (\tau_1 + \tau_2) \Rightarrow \Gamma']\!]\ (\mathbf{Inl}\ v)\ \rho & = \mathsf{fail}_{\mathcal{M}} \\
[\![\Gamma \vdash (p_1, p_2) : (\tau_1 \times \tau_2) \Rightarrow \Gamma_3]\!]\ (v_1, v_2)\ \rho = & ([\![\Gamma \vdash p_1 : \tau_1 \Rightarrow \Gamma_1]\!]\ v_1\ \rho) \star_{\mathcal{M}} \lambda\rho_1. \\
 & \quad ([\![\Gamma_1 \vdash p_2 : \tau_2 \Rightarrow \Gamma_2]\!]\ v_2\ \rho_1)
\end{array}
$$

The meaning of patterns is defined by induction on the derivations of the pattern inference judgment $\Gamma \vdash p : \tau \Rightarrow \Gamma'$. The meaning is a function that deconstructs a value of type $[\![\tau]\!]$ and produces a runtime environment transformer that either augments the runtime environment with bindings for variables in the pattern p. Note that deconstructing a sum value, if there is a mismatch in the injections between the pattern and the value, may result in failure. Hence the type

$$[\![\Gamma \vdash p : \tau \Rightarrow \Gamma']\!] : [\![\tau]\!] \to [\![\Gamma]\!] \to \mathcal{M}\ [\![\Gamma']\!]$$

Now, let us consider the semantics of functions with pattern matching:

$$[\![\Gamma \vdash \lambda p.\, e : \tau_1 \to \tau_2]\!]\ \rho \quad = \lambda v.([\![\Gamma \vdash p : \tau_1 \Rightarrow \Gamma']\!]\ v\ \rho) \star_{\mathcal{M}} \lambda\rho'.([\![\Gamma' \vdash e : \tau_2]\!]\ \rho')$$

The meaning of functions consists of two parts: first, meaning of pattern deconstructs the function argument. If this computation succeeds, a new runtime environment ρ' is constructed and the body of the function is evaluated in this new environment.

Finally, we consider the meaning of case expressions.

$$
\begin{array}{lll}
[\![\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \overline{p_n \to e_n} : \tau']\!]\ \rho = & & [\![\Gamma \vdash e : \tau_1]\!] \star_{\mathcal{M}} \lambda v. \\
 & & [\![\Gamma \vdash p_1 : \tau \Rightarrow \Gamma_1]\!]\ v\ \rho \star_{\mathcal{M}} \lambda\rho_1.[\![\Gamma_1 \vdash e_1 : \tau']\!]\ \rho_1 \\
 & [\![& [\![\Gamma \vdash p_2 : \tau \Rightarrow \Gamma_2]\!]\ v\ \rho \star_{\mathcal{M}} \lambda\rho_2.[\![\Gamma_2 \vdash e_2 : \tau']\!]\ \rho_2 \\
 & [\![& \qquad\qquad\qquad\qquad \cdots \quad \cdots \quad \cdots \\
 & [\![& [\![\Gamma \vdash p_n : \tau \Rightarrow \Gamma_n]\!]\ v\ \rho \star_{\mathcal{M}} \lambda\rho_n.[\![\Gamma_n \vdash e_n : \tau']\!]\ \rho_n
\end{array}
$$

5.3.3 Implementation of L_1 with Patterns

The data-type `Pat t gin gout` corresponds to the well-typedness judgments on patterns $\Gamma_{\mathsf{in}} \vdash p : \tau \Rightarrow \Gamma_{\mathsf{out}}$.

```
data Pat t gin gout =
   PVar  (Equal gout (gin,t))
 | ∀αβ. PInl (Pat α gin gout) (Equal t (Either α β))
 | ∀αβ. PInr (Pat β gin gout) (Equal t  (Either α β))
 | ∀αβγ. PPair (Pat α gin γ) (Pat β γ  gout) (Equal t (α,β))
```

Note that we omit the actual encodings of patterns in the type `Pat` since it is unnecessary to the development presented here. The case for variable-binding patterns `PVar` carries the proof that the type of the target type assignment `gout` is equal to the source type assignment `gin` paired with the type of the pattern itself (`Equal gout (gin,t)`). The left injection pattern `PInl` takes as its argument the proof of a pattern judgment `Pat α gin gout`, together with the proof that `t` equals `Either α β`. The most interesting case is the pair pattern. Its first argument is a proof of the pattern judgment `Pat α gin β`. The target type assignment of the first argument, γ, is then given as a source type assignment to the second argument `Pat β γ gout`, thus imposing a sequence on type assignment extension for pairs. Finally, this constructor also needs a proof that the type of the pattern `t` equals `(α,β)`.

Below, we give the definitions of the smart constructors for building proofs of pattern judgments.

```
pvar :: Pat a b (b,a)
pvar = PVar refl

pinl :: Pat a b c→Pat (Either a d) b c
pinl pat = PInl pat refl

pinr :: Pat a b c→Pat (Either d a) b c
pinr pat = PInr pat refl

ppair :: Pat a b c→Pat d c e→Pat (a,d) b e
ppair pat1 pat2  = PPair pat1 pat2 refl
```

The next step is to extend the definition of expressions to work with patterns.

```
data Exp e t
  = Lit Int (Equal t Int)
  | V (Var e t)
  | ∀αβγ. Abs (Pat α e γ) (Exp γ β) (Equal t (α→β))
  | ∀α. App (Exp e (α→t)) (Exp e α)
  | ∀αβ. Pair (Exp e α) (Exp e β) (Equal t (α,β))

```

```
8  abs :: Pat a b c → Exp c d → Exp b (a → d)
9  abs p e = Abs p e refl
```

The only change from the previous definition (Figure 5.2) is the λ-abstraction case (Line 4): the abstraction constructor takes as its first argument a pattern judgment which, given an argument of the domain type α, produces an extended type assignment γ. Then, the judgment for the body of the abstraction is typed under the the extended type assignment $\gamma$with the codomain type β. Note, that with the introduction of pair patterns, we have dispensed with the need for separate elimination constructs (`Pi1` and `Pi2`) for the product types.

Example We list an example well-typed term with patterns:

```
-- swap = λ(•, •).(Var 0, Var 1)
   swap :: Exp a ((b,c) → (c,b))
   swap =  abs (ppair pvar pvar) (pair (V z) (V (s z)))
```

The function swap uses the pattern $(\bullet, \bullet)$ to deconstruct a pair, and returns a pair with the order of its elements reversed. Note that the variable index zero, `V z`, refers to the rightmost variable in the pattern.

Case expressions Case expressions are used to eliminate sum types. We extend the typing judgment for expressions with the constructor `Case`:

```
data Exp e t =
  . . .     . . .     . . .      . . .
  | ∀α. ECase (Exp e α) [Match e α t]

data Match e t' t = forall e'. Match (Pat t' e  e') (Exp e' t)
```

A case expression consists of a discriminated expression of type `Exp e` α, and a list of *matches*. The data-type `Match` is a ternary type constructor: its first argument is the type assignment `e`; its second argument is the type of the discriminated expression `t'`; its third argument is the result type of the match `t`. Since each pattern in a match may bind a different number of variables, the type assignment that the right hand side expression of each of the matches in a case may be different. Thus, an existential type is introduced in the definition of the matches. A match is a pair of a pattern and an expression, where for each match there exists an output type assignment `e'` produced by that pattern, in which the expression is typed.

An Interpreter for L_1^+

There are a number of design choices to take when implementing the interpreter for the language with pattern-matching. The first is how to handle local and global failure. For the interpreter we will present here we have opted for the following:

1. Global failure is modeled by a non-terminating Haskell computation. This diverges somewhat from the set-theoretic model we have outlined above, but it makes our definitions more concise.

2. Local failure is modeled by computations in the Haskell `Maybe` monad, as outlined in Section 5.3.2. In case expressions, after alternatives to the local failures are exhausted, global failure is raised.

The first step is to implement the evaluation function for patterns.

```
1  fatbar :: Maybe a → Maybe a → Maybe a
2  fatbar (Just x) e = Just x
3  fatbar Nothing e = e
4
5  evalPat :: (Pat t ein eout) → t → (ein → Maybe eout)
6  evalPat (PVar p) v    = \e → return (b2a p (e,v))
7  evalPat (PInl pt p) v = \e → case a2b p v of
8                                 Left x → evalPat2 pt x e
9                                 Right _ → Nothing
10 evalPat (PInr pt  p) v = \e → case a2b p v of
11                                  Left _ → Nothing
12                                  Right x → evalPat2 pt x e
13 evalPat (PPair pat1 pat2 p) v = \e →
14  let v' = a2b p v
15  in do { e' <- evalPat2 pat1 (fst v') e
16        ; evalPat2 pat2 (snd v') e'}
```

The function `evalPat` takes a proof of the pattern judgment `Pat t ein eout`, a value of type `t`, and returns an *environment transformer* function `ein → Maybe eout`, where the `Maybe` type in the co-domain indicates that pattern matching may fail (local failure).

The case for variables is trivial: given a value `v`, the environment transformer simply adds the value `v` onto the initial environment.

The case for Inl patterns is more interesting. First, the value `v` is discriminated to determine whether it is the left or right injection of a sum. If it is the left injection, `evalPat` recursively deconstructs the sub-pattern with the projection of the value. If, however, the value has the form of the right injection, failure is symoked using the non-proper morphism `fail`. The case for the right injection pattern is symmetric.

Finally, for pair patterns, `evalPat` first evaluates the left sub-pattern with the left element of the pair value. Then, the right sub-pattern is recursively matched. The environment is threaded through from the results of the left to the input of the right pattern match.

Having defined `evalPat`, we are ready to give semantics of λ-abstractions with patterns and the case expressions:

```
1  eval :: (Exp e t) → e → t
2  . . .    . . .    . . .      . . .
3  eval (Abs pat exp p) env = b2a p
4    (\x →
5      case (evalPat pat x env) of
6        Just env' → eval exp env'
7        Nothing → error "Pattern match failure in abstraction!")
8  eval (Case e branches) env =
9    case (evalCase (eval e env) branches env) of
10    Just v → v
11    Nothing → error "Pattern match failure"
12
13 evalCase :: t1 → [Match e t1 t2] → e → Maybe t2
14 evalCase val [] env = fail
15 evalCase val ((Match pat branch):rest) env =
16   (do { e' <- evalPat pat val env
17       ; return (eval branch e') })
18   `fatbar`
19   (evalCase val rest env)
```

The case of `eval` for abstraction (line 3-7) creates a function value whose argument, `x`, is passed to `evalPat` in order to create an extended runtime environment `env'`. In case of failure of `evalPat`, an error is raised. If the pattern matching succeeds, the body of the function is evaluated in the augmented runtime environment `env'`.

The case of `eval` for the case expressions first evaluates the expression to be discriminated, and passes the resulting value to the function `evalCase`. If `evalCase` succeeds, its value is returned as the final result. In case of failure of `evalCase`, a pattern matching error (global failure) is raised.

The function `evalCase` (lines 13-19) performs the evaluation of a case expression. If there are no matches left to examine (line 14), failure is raised. Otherwise, for each of the matches, the pattern is evaluated with `evalPat` against the value of the discriminated expression. If the pattern match succeeds, the augmented environment is passed on to `eval` of the right hand side of the match. If a local failure occurs along the way, `evalCase` re-tries with the next match.

Example As a more comprehensive example, we implement the factorial function. This function uses a few more syntax building combinators than has been introduced in the previous text. These are additions to L_1^+ that support recursive definitions (`fix`), arithmetic and integer comparison operations. Their signatures are given below:

```
1  data Exp e t = . . .
2               | Fix (Exp (e,t) t)
3
4  fix :: Exp (e,t) t → Exp e t
5  fix e = Fix e
6
7  eval (Fix e) env = eval e (env, eval (Fix e) env)
8
9  lte :: Exp e Int → Exp e Int → Exp e Bool
10 times,minus :: Exp e Int → Exp e Int → Exp e Int
11
12 fact2 :: Ext e (Int → Int)
13 fact2 = fix
14    (lam pvar
15      (ecase (lte (V z) (lit 0))
16             [ Match  (pinl pvar) (lit 1)
17             , Match  (pinr pvar) body   ]))
18            where body = times (V (s z)) rcall
19                  rcall = (V (s (s z))) ‘app‘  (minus (V (s z)) (lit 1))
```

Note that we use lines and arrows to connect a use site of a variable with its binding site. Note also that `fix` binds a variable which is used in the recursive call to the factorial.

5.4 Staging

The technique for encoding and interpreting languages presented in the previous sections may at first appear untagged. The interpreter function `eval` has the type `Exp e t` → `e` → `t`: instead of injecting all possible types of its result values into a single value domain, the interpreter returns “untagged” values: integers, functions, and so on[6]. However, instead of tagging with injections into the universal domain, these interpreters exhibit another form of tagging, as can be recalled from the following part of the definition of `eval`:

[6] One should note that a number of programming language features come together to make this possible. The use of equality types has already been explained in considerable detail. Furthermore, parametric polymorphism and polymorphic recursion allow us to type functions like `eval`.

```
eval :: (Exp e t)→e→t
eval (Lit i p) env = b2a p i
eval (Pair e1 e2 p) env = b2a p (eval e1 env, eval e2 env)
```

Note that the boxed casting operations in fact play the same role as injections and projections of the universal value domains in more traditional implementations of interpreters in Haskell. In this section, however, we will point to a crucial difference between tagging/untagging operations with an universal value domain and the casting and equality operators we use. This distinction becomes visible and practically useful only when we add *staging* to the meta-language.

5.4.1 Staging: Interpretive and Tagging Overhead

We will first make a small digression to introduce and motivate the notion and techniques of `staging`. Consider the following interpreter for L_1^+. We use typing judgment as usual, but the range of the eval is a universal domain of values (`V`) which is a sum of functional values (tag `VF`), integers (tag `VI`), pairs (tag `VP`), and tagged sums (tag `VS`). The interpreter `eval0` contains uses of tagging and untagging operations (i.e., the operations which inject or project into/out of the universal value domain), which are highlighted.

```
data V = VF (V→V) | VI Int | VP V V | VS (Either V V)
unVF (VF f) = f

eval0 :: Exp e t→[Val]→Val
eval0 (Lit i _) env = VI i
eval0 (V var) env = evalVar0 var env
eval0 (App f x) env = unVF (eval0 f env) (eval0 x env)
eval0 (Abs pat e _) env =
      VF(\v→eval0 e (unJust(evalPat0 pat v env)))

evalVar0 :: Var e t -> [V] -> V
evalVar0 (Z _) (v:vs) = v
evalVar0 (S s _) (v:vs) = evalVar0 s vs

evalPat0 :: Pat t ein eout -> V -> [V] -> Maybe [V]
evalPat0 (PVar _) v env = return (v:env)
... ... ...
```

An examination of the interpreter `eval0` reveals two sources of inefficiency:

Interpretive overhead. Interpretive overhead [67] is the main reason why interpreted programs are as a rule less efficient than compiled programs. The overhead comes from the fact that the interpreter must spend considerable computation time and resources to analyze and interpret a program *at runtime* of the program it is interpreting. For example, the function `eval0` calls itself recursively in the body of a `VF` value when it interprets an `Abs` expression. Moreover, these recursive calls to `eval0` are *latent:* they are not symoked until the function value `VF f` is untagged and applied at run-time. If the functional value is applied many times, the latent recursive calls will be performed each time.

A more efficient implementation can be obtained by specializing the interpreter with respect a given object program (the first Futamura projection [38, 67]): this in effect *unfolds* the interpreter "statically," at a stage earlier than the actual execution of the program being interpreted, thus removing from the runtime execution of the program all the operations on its source syntax performed by the interpreter. This means, among other things, that latent recursion present in the `VF` case can be removed by "evaluating under the lambda" of the tag `VF`. For example, instead a value

```
VF(\v→eval0 (V z) (v:env))
```

we obtain the equivalent, but more efficient

```
VF (\v→ v)
```

Traditionally, partial evaluation has been used to perform this kind of specialization of interpreters. Meta-programming by *staging* offers a particularly elegant way of removing this interpretive overhead (e.g., [117]).

Following MetaML [137, 136], we will introduce into our meta-language a type of `code`, (here written $\bigcirc t$, taking the syntax from Davies [30, 29]) which indicates "computation of `t` deferred to the next computational stage." An introduction construct for this code type are the *code brackets* $\langle$`e`$\rangle$, which delay the expression `e` of type `t`, obtaining a value of type $\bigcirc$`t`. Code can be "spliced" into a larger code context by the *escape* expression `~e`. When an escape expression appears inside code brackets, the escaped expression is computed at the earlier computational stage. The results of this computation (which itself must be a code value) is then "spliced" at the same spot in the delayed context where the escape had first appeared.

We consider adding staging constructs to Haskell as a conservative extension relatively uncontroversial. Combining staging constructs with a call-by-name language should be no more difficult than combining them with a call-by-value one [129]. We have implemented an interpreter for a Haskell-like language with staging [124], in which the subsequent program examples are written.

We will now stage the example interpreter `eval0`, obtaining a two-stage version, `eval0S`. The execution of the function `eval0S` is divided into two distinct computational stages: in the first stage, the

interpreter is unfolded over a particular expression, performing all the interpretive operations on the syntax of the program itself; in the second stage (properly a run-time stage of the interpreted expression), only computation pertaining to the object program on which the interpreter was specialized remains.

```
eval0S :: Exp e t → ◯[Val] → ◯Val
eval0S (Lit i _) env = ⟨VI i⟩
eval0S (V var) env = evalVar0S var env
eval0S (App f x) env = ⟨unVF ~(eval0S f env) ~(eval0S x env)⟩
eval0S (Abs pat e _) env =
   ⟨VF(\v → ~(eval0S e ⟨unJust ~(evalPat0S pat ⟨v⟩ env)⟩))

evalVar0S :: Var e t → [◯V] → ◯V
evalVar0S (Z _) env = ⟨head env⟩
evalVar0S (S s _) env = evalVar0S s ⟨tail ~env⟩

evalPat0S :: Pat t ein eout → V → [◯V] → ◯(Maybe [V])
evalPat0S (PVar _) v env = ⟨return (v:env)⟩
... ... ...
```

Applying the function `eval0S` to an example expression yields the following result:

```
ex1 =  eval2 (Abs (Abs (App (Var 0) (Var 1)))) []
--⟨VF (\x → VF(\y → unVF (head [y,x]) (head (tail [y,x]))))⟩

v1 = run ex1
v1 :: V
```

MetaML also has a `run` operation which takes an expression of type ◯`t` and runs the delayed computation now, yielding a value of type `t`. It is important to note that `run ex1` returns a `V` from whose evaluation all recursive calls to `eval0S` have been removed: even though it is easily provable that in MetaML [129] `(eval0 e [])` is semantically equivalent to `(run (eval0S e []))`, the latter expression executes potentially considerably faster then the first (see [62, 16] for some experimental measurements).

Tagging overhead. Another kind of overhead introduced into interpreters is *tagging overhead* (for a detailed explanation see Section 2.1.1). Tagging overhead occurs in certain situations when both the meta-language and the object-language are strongly typed, but the type system of the meta-language forces the programmer to "sum up" the values of the object-language programs purely in order to make the interpreter

type-check [102]. If we only consider interpreting well-typed object language programs, these tags are unnecessary – the strong typing of the object language ensures that no tag mismatch occurs at runtime. This is the case with the interpreter `eval0` given above, since evaluating proofs judgments restricts the function to evaluating only well-typed L_1^+ expressions. When such an interpreter is staged, the tagging and untagging operations are inherited by the residual program.

For example, the residual program first shown above has three tagging operations (shown boxed):

⟨ VF (\x→ VF (\y→ unVF (head [y,x]) (head (tail [y,x])))) ⟩

When interpreting large programs, these tags can proliferate and cause considerable run-time performance penalty [133].

5.4.2 Staging the Interpreter for L_1^+

We will proceed with staging of the interpreter for L_1^+ in a couple of steps. First, we will make the most straightforward (naïve) staging modification to the interpreter we have already presented. Then, we will discuss how certain *binding time improvements* [67] can be made to the original interpreter to make staging even more efficient.

First Attempt

The simplest way of staging an interpreter is to begin with the text of the original (non-staged) interpreter, and simply add staging annotations to it, separating the interpreter into two phases: the static (staging time) and dynamic (run-time) phase. In this operation we are guided by types: we shall add a single circle type to the types of values we expect to be performed in the dynamic phase.

Thus, the type of the `eval` function is changed from `(Exp e t)` → `e` → `t` to `(Exp e t)` → ◯`e` → ◯`t`, meaning that the runtime environment binding values to variables, and the value returned by the interpreter are dynamic. The source L_1 program itself (`Exp e t`) remains static.

We now examine the annotations and changes that need to be made to the definition of `eval`.

```
1 evalS :: Exp e t → ◯e → ◯t
2 evalS (Lit i p) env = castTa (sym p) ⟨i⟩
3 evalS (V v) env = evalVarS v env
4 evalS (Abs pat body p) env = castTa (sym p)
5    ⟨\x → (let env2 = unJust ~(evalPatS pat ⟨x⟩ env)
6            in  ~(evalS body ⟨ env2 ⟩))⟩
7 evalS (App e1 e2) env = ⟨~(evalS e1 env) ~(evalS e2 env)⟩
8
```

```
9  evalVarS :: Var e t → ◯e → ◯t
10 evalVarS (Z p) env =  ⟨snd ~(castTa p env)⟩
11 evalVarS (S v p) env =  ⟨ let env2 = fst ~(castTa p env)
12                           in ~(evalVarS v ⟨env2⟩)⟩
13
14 evalPatS :: Pat t ein eout → ◯ t → ◯ ein → ◯(Maybe eout)
15 evalPatS (PVar  p) v ein = ⟨Just ~(castTa (sym p) ⟨(~ein,~v) ⟩)⟩
16 evalPatS (PInl pt p) v ein =
17  ⟨ case ~(castTa p v) of
18       Left x →~(evalPatS pt ⟨x⟩ ein)
19       Right _ →Nothing⟩
20 evalPatS (PInr pt p) v ein =
21  ⟨case ~(castTa p v) of
22       Left _ → Nothing
23       Right x → ~(evalPatS pt ⟨x⟩ ein)⟩
24 evalPatS (PPair pat1 pat2 p) v ein =
25  ⟨let (v1,v2) = ~(castTa p v)
26     in  do { eout1 <- ~(evalPatS pat1 ⟨v1⟩ ein)
27            ; ~(evalPatS pat2 ⟨v2⟩ ⟨eout1⟩) }⟩
```

The simplest case is evaluating integer literals:

```
(2) evalS (Lit i p) env = castTa (sym p) ⟨i⟩
```

The integer value ⟨`i`⟩ is returned in the next stage. Note that the casting operation is changed from `b2a p :: Int → t` to `castTa (sym p)::◯Int → ◯t` – which reflects the fact that cast must be "pushed through" the ◯ type constructor. Similar changes to casting operations to make them work in a code context are made throughout the interpreter.

Next, we examine the variable case:

```
(9) evalVarS :: Var e t → ◯e → ◯t
(10) evalVarS (Z p) env =  ⟨snd ~(castTa p env)⟩
(11) evalVarS (S v p) env =  ⟨ let env2 = fst ~(castTa p env)
                               in ~(evalVarS v ⟨env2⟩)⟩
```

The auxiliary function `evalVarS` is similarly annotated to ensure that the environment is projected from at runtime of the object program. Thus, evaluating variable `(s z)` with some environment ⟨`e`⟩ results in ⟨`snd (fst e)`⟩. Note that projection of the appropriate value from the environment is thus completely delayed to the runtime.

Example

Let us now consider staging a sample L_1 program.

```
ex1 :: Exp a (Int→Int→(Int,Int))
ex1 = abs (abs (pair (V z) (V (s z))))

- evalS ex1 ⟨()⟩
⟨\x→\y→(snd ((),x),y), fst (snd ((),x),y))⟩
```

Two things should be noted. First, much of the interpretive overhead has been removed from the resulting expression.

```
⟨\x→\y→(snd ((),x),y), fst (snd ((),x),y))⟩
```

However, one small piece of this overhead remains: whenever a variable is evaluated, it is looked up in the environment dynamically. This is too dynamic, since the actual position in the runtime environment is known *statically* and does not change for each variable.[7] Recognizing this fact and changing our implementation to take advantage of it constitutes a *binding time improvement*, which we shall discuss later.

Second, *all* tagging overhead has been removed from the resulting code. This is a significant improvement over earlier implementations of staged interpreters (e.g., [117]). It was made possible by a careful use of equality operators and casting: since code is just another type constructor, we were able to cast a type "through" code – allowing us to perform the actual casting at an earlier stage. This behavior is very reminiscent of *tag elimination* [132, 133], where a separate stage (between static and runtime stages) is used to perform the elimination of tagged values in residual code of a staged interpreter. Here, the rôle of this special tag elimination stage is played by stage 0, while stage 1 becomes the run-time stage for interpreted programs.

Binding Time Improvements

The process of (slightly) changing the interpreter to make it more amenable to staging is known as *binding time improvement* [67]. In the remainder of this section, we will make two binding time improvements to the staged interpreter for L_1^+ with the goal of removing even more interpretive overhead, especially the dynamic lookup mentioned above.

1. **Partially static environments.** What the previous staged interpreter fails to take advantage of is the fact that the runtime environment is *partially static*. Namely, while the values in the environment are

[7] In other words, the environment in this interpreter is *partially static*.

not known until stage one, the actual *shape* of the environment is known statically and depends only on the structure of the term being interpreted. We should be able to do away with the dynamic lookup of values in the runtime environment. The resulting interpreter should produce residual code for the above example that looks like this:

$\langle$`\x → \f → f x`$\rangle$

2. **Pattern matching and control flow.** Pattern matching as presented in the semantics above relies on the failure monad, and the `fatbar` operator to propagate pattern matching failure. This makes residual code rather complicated and less efficient. A standard technique in staging is to rewrite such code in *continuation passing style*. Instead of propagating failure with the monad, we will simply rewrite our pattern matching function `evalPat` to accept a success-and-failure continuation. The residual code produced by this interpreter is much cleaner and easier to read, implementing cases in L_1 by cases in the residual program.

Partially Static Environments. Recall that environments in the previous definitions of the interpreter are dynamic nested pairs of the form $\langle$ `((...,v2),v1)`$\rangle$. The corresponding partially static environment is a set of static nested pairs, in which each second element is a dynamic value: `((...,`$\langle$`v2`$\rangle$`,`$\langle$`v1`$\rangle$`)`. This relationship between environment types and the corresponding partially static environments is encoded by the following data-type:

```
data PSE e
  =      INIT ◯e
  | ∀αβ. EXT (PSE α) ◯β (Equal e (α,β))

-- smart constructor
ext :: PSE a → ◯b → PSE (a,b)
ext e t = EXT e t refl
```

A partially static environment (hence, a PSE) can either be completely dynamic (`INIT`), or it can be an environment extended by a dynamic value. The equality proof argument ensures that the type argument `e` is identical in form to the form of type assignment index of judgments (the `e` in `(Exp e t)` and `(Var e t)`). Now, we can give a new type to the interpreter, as follows:

```
eval2S    :: Exp e t → (PSE e) → ◯t
evalVar2S :: Var e t → (PSE e) → ◯t
```

The interpreter now takes a well-typed expression `(Exp e t)`, and a partially static environment `(PSE e)`, and produces a delayed result of type $\bigcirc$`t`. The largest change is in the evaluation function for variables, `evalVar2S`:

```
evalVar2S :: Var e t → (PSE e) → ◯t
evalVar2S (Z p) (EXT _ b p2) = castTa prf b
  where (_,prf) = pairParts (trans (sym p2) p)
evalVar2S (S s p) (EXT e _ p2) = evalVar2S s (castTa prf e)
  where (prf,_) =  (pairParts (trans (sym p2) p))
```

The base case takes a derivation of the typing judgment for variable zero, which contains the equality proof `p::Equal e (`α`,t)`. Its second argument is a PSE, with `b::`$\bigcirc\beta$, and the proof `p2` of type `Equal e (`α`,`β`)`. The main work is performed by constructing the proof `prf`, which shows that β is equal to `t`. A simple cast then converts the value `b` from the type`(`$\bigcirc\beta$`)` to `(`$\bigcirc$`t)`. Note that the definition of `prf` uses the product equality axiom `pairParts`.

The inductive case is similar. The pair equality axiom is again used to obtain a proof object and cast the sub-environment so that the recursive call to `evalVar2S` is well typed.

The functionality of `evalVarS` can also be retained by simply providing two additional cases for `evalVar2S`, i.e., when the PSE is of the form `(INIT dynenv)`.

```
evalVar2S (S s p) (INIT env) = evalVar2S s (INIT ⟨fst ~(castTa p env)⟩)
evalVar2S (Z p) (INIT env) = ⟨snd (~(castTa p env))⟩
```

Pattern Matching and Continuations. We have seen how PSE's are used by the new interpreter. It remains yet to see how those environments are extended. Rewriting the function `evalPat2S` in continuation passing style is not difficult. We start by giving it a new type:

```
evalPat2S :: Pat t ein eout →
             ◯t → (PSE ein) →
             (Maybe (PSE eout) → ◯ans) → ◯ans
```

The function `evalPat2S` takes a pattern judgment `(Pat t ein eout)`, a delayed value of type `t`, an input PSE of type `ein`, and a *continuation* function. The continuation takes as its argument a maybe type which is either a new, augmented PSE `eout`, or `Nothing` and returns a piece of code of some answer type `ans`. When given the `(Just ein)` argument, the continuation constructs the answer for the case in

which the pattern matching succeeds. When given `Nothing`, the continuation constructs the code for the case in which the pattern matching fails.

```
evalPat2S (PVar p) v ein k = k (Just (castTa (sym p) (ext ein v)))
```

The variable case always succeeds. Therefore, the input PTE is extended by the value (`ext ein v`), and passed to the continuation as success (`Just`).

```
evalPat2S (PInl pt p) v ein k =
  ⟨case ~(castTa p v) of
        Left x → ~(evalPat2S pt ⟨x⟩ ein k)
        Right x → ~(k Nothing) ⟩
```

The `PInl` case (lines 38-39) creates a piece of code which first analyzes the input value `v`, generating a `case` expression with two branches. The first branch is generated for the case where value is of the form `(Left x)`. Its body is generated by `evalPat2S` which calls itself recursively on the sub-pattern `pt` and input value ⟨`x`⟩ without modifying the continuation `k`. The other branch, however, concerns the situation where the input value is of the form `(Right x)`, i.e., a mismatch has occurred. The body of this branch is generated by the continuation `k`, symoked with failure, `~(k Nothing)`.

Finally, we examine the case for pair patterns (lines 43-47).

```
evalPat2S (PPair pt1 pt2 p) v ein k =
  ⟨case ~(castTa p v) of
       (v1,v2) → ~(evalPat2S pt1 ⟨v1⟩ ein (h ⟨v2⟩)) ⟩
 where h n Nothing = k Nothing
       h n (Just eout1) = evalPat2S pt2 n eout1 k
```

Given a pair pattern with sub-patterns `pt1`, and `pt2`, and an input value `v`, the input value pair is first deconstructed into its elements `v1` and `v2`. Then, `evalPat2S` calls itself recursively with the left sub-pattern `pt1`, the value ⟨`v1`⟩, the input environment `ein`, and, most importantly, the enlarged continuation `h` ⟨`v2`⟩. The continuation `h` ⟨`v2`⟩ (lines 46-47) discriminates against its argument:

1. If it is `Nothing`, then a previous pattern match must have failed and it symokes the initial continuation `k` with `Nothing` to propagate the failure.

2. If the previous pattern matching has succeeded with some new augmented environment `eout1`, it symokes `evalPat2S` recursively with the right-hand side patterns and values, giving it the new environment as its input, and the initial continuation `k`.

Putting It All Together. The full implementation of the binding time improved interpreter for L_1^+ is given in Figure 5.7 on page 141. Combining all the improvements shown above, we can return to `eval2S`. Consider, for example, the case for λ-abstraction (lines 14-16). This case constructs a piece of code that is a function $\langle$`\x → ...`$\rangle$. The body of this function is constructed by a call to `evalPat2S` which is given the pattern, the discriminated value $\langle$`x`$\rangle$, the current environment and the continuation `h`. The continuation `h` generates the body of the λ-abstraction using the enlarged environment constructed by `evalPat2S` in cases of success, and error raising code `error "failure"` in case the pattern matching fails.

For example, when run with the input program λInl $\bullet$. (Var 0), the staged interpreter `eval2S` returns the following code:

```
⟨\x → case x of (Left y) → y
                (Right z) → error "failure"⟩
:: code ((Either a b) → a)
```

It is also worth noting that if the pattern abstracted over by an L_1 abstraction does not contain sums, the failure portion of the continuation is never symoked, and no case expressions are generated. For example, the input program $\lambda \bullet$ .Var 0:

```
⟨\x → x⟩ :: (forall a . code (a → a))
```

Let us also consider how case expressions are defined: cases are constructed using the auxiliary function `evalCase2S` (lines 20-24).

```
eval2S (ECase e matches) env =
  ⟨let value = ~(eval2S e env)
     in ~(evalCase2S ⟨value⟩ matches env)⟩

evalCase2S :: ◯t1 → [EE e t1 t2] → PSE e → ◯t2
evalCase2S val [] env = ⟨error "failure"⟩
evalCase2S val ((EE (pat,branch)) : rest) env =
 evalPat2S pat val env h
   where h (Nothing) = evalCase2S val rest env
         h (Just env2) = eval2S branch env2
```

First, code is constructed for the discriminated expression, and bound to the variable `value`. Then, `evalCase2S` is called to match all the branches of the cases against $\langle$`value`$\rangle$. This trick is used to

prevent code duplication between individual matches. The evaluation of each match proceeds just as with λ-abstraction. The only difference is that in case of failure, the continuation `h` symokes `evalCase2S` recursively to construct further branches for all alternatives.

We show the code generated by `eval2S` for the expression

$$\begin{array}{lll} \lambda \bullet .\mathsf{case} & \mathsf{Var}\ 0\ \mathsf{of} \\ & \mathsf{Inl}\ \bullet \rightarrow \mathsf{Var}\ 0 \\ & \mathsf{Inr}\ \bullet \rightarrow \mathsf{Var}\ 0 \end{array}$$

```
⟨ \x→let v = x
        in case v of (Left y)→y
                     (Right _)→
                       case v of (Left  _)→error "fail"
                                 (Right z)→z  ⟩
```

5.5 Conclusions

In previous chapters we have proposed and elaborated on a technique for implementation of strongly typed object languages. Essential to this technique are certain properties of the object language, such as being strongly typed; these properties are used to justify producing interpreters which are efficient and reliable by construction. In particular, we have used dependent types to encode inductive sets of only *well-typed* terms. Interpreters can be defined over these well-typed terms to avoid tagging overhead, and staged to avoid interpretive overhead.

In this chapter, we have explored the extent to which similar techniques can be adapted in the setting of the more popular programming language Haskell. The motivation for this is twofold. First, we wish to explore the power and flexibility of Haskell-like type systems in order to understand its potential for meta-programming. The second reason is pragmatic: although the meta-programming system with dependent types has many useful theoretical properties, such systems have yet to develop a wider user base, and is thus liable to gain wider acceptance.

5.5.1 Computational Language vs. Specification Language

To implement object languages (interpreters, compilers, type-checkers, static analysis tools, and so on), one needs a meta-language. The meta-programming framework we have developed requires the meta-language

to be *typed*. In such a typed language we can distinguish between a *computational stratum* of the meta-language which describes the programs that are executed at runtime, and a static *specification stratum*, which is used to specify properties of the programs in the computational stratum. In functional languages the distinction between these is rather simple: programs that result in some value-yielding computation at runtime are the computational stratum, while types of these programs whose validity is checked statically (at type-checking time) are the specification stratum. The distinction between these two coincides with the usual separation between static and dynamic phases of a program execution.

This *phase distinction* [17] is often difficult to maintain in programming languages with dependent types since type-checking (static phase) often requires evaluation (dynamic phase) because types can depend on dynamic values. In MetaD [102] and FLINT [116], this distinction is maintained by an elaborate stratification: the language is explicitly divided into a *computational language* whose expressions are classified by a type system (*specification language*) which itself is a highly expressive language (a version of the Calculus of Inductive Constructions [24, 22]). The specification language is expressive enough that interesting domain values that exist at runtime (such as integers) can also be represented at the level of types. Logical propositions are then also implemented at the level of kinds, proofs of these propositions being types. *Singleton types* [57] are used to force a correspondence between runtime-values and their representations at type level.

In our Haskell implementation, this complex structure must be mapped into the only two levels available: runtime Haskell programs and static Haskell types (and type constructors, and so on). We summarize the main correspondences.

Domain value types. These Haskell types that are conceptually intended to represent runtime values *at the level of types.* They correspond to the elements of the inductive kind `Nat` in FLINT [116]. One difference between our encoding of domain-value types and the inductive kinds in MetaD and FLINT is that we have no way of enforcing *a priori* the well-formedness of such domain value types – rather, in our Haskell implementation, they are like terms upon whom structure must be imposed by a disciplined use of these terms.

Well-formedness judgments. These play a dual role in the Haskell implementation. First, they are there to impose a structure on *domain value types*: the type `IsNat` from Section 4.3 is a particularly good example of this. For example, the type `S (S Z)` represents the natural number 2, but the type `S (S (String → Int))` should be excluded from consideration as a valid representation in the domain of integers. Type constructor `IsNat` ensures that any argument type given to it is well-formed; by requiring `IsNat` types as arguments for functions, the user can ensure that only well-formed domain value naturals are used in types of her programs.

Well-formedness judgments as singleton types. Another important consideration is to connect runtime values to the domain value types that describe them. A standard way that has been proposed to deal with this is to introduce a type constructor `snat : Nat` $\rightarrow$ `*` (where `Nat` is the inductive kind of natural numbers) such that

> given a type term n of kind `Nat`, if a computation value v has type `snat` n, then v denotes the natural number represented by n. [116].

A good example that illustrates this connection in our Haskell implementations is the addition of numbers (Section 4.3). The way we speak about addition between domain value types at type-checking time is by the type constructor `PlusRel m n z`: a value of the type `PlusRel m n z` is a proof that `z` equals `m` plus `n`. However, this property must ultimately be connected to some *runtime* value and a function that performs addition at runtime. A number of systems establish this connection between runtime values and their representations at type level through singleton types [57, 148, 116].

In our Haskell implementation, however, the role of `snat` is played by certain well-formedness judgments (e.g., `IsNat`). The MetaD and FLINT the system ensures that this representation is correct by construction (a meta-theorem guarantees that `snat` *adequately* represents runtime natural number values), in our framework, the user must ensure that the well-formedness judgment corresponds to the objects that are modeled closely enough (usually a 1-to-1 correspondence) so that the proofs (or derivations) of these well-formedness judgments can be used *as a representation of the objects themselves.*

In most interesting cases (inductively defined sets, such as the set of natural numbers), this is easily established (e.g., `IsNat n` $\cong \mathbb{N}$, for any n). Still, it is important to emphasize that the burden of establishing this correspondence falls upon the programmer, and that there seems to be no way to prove this adequacy *within the system itself.*

5.5.2 Staging

We have also shown that staging can be successfully combined with equality-proof based implementations of programming languages. In particular, the combination of staging and equality proofs allows us to write staged interpreters from which *tagging overhead* has been removed *by construction.*

```
data PSE e =      INIT ◯e
           | ∀αβ. EXT (PSE α) (◯β) (Equal e (α,β))

ext :: PSE e→◯t→PSE (e,t)
ext  e t = EXT e t refl

eval2S :: Exp e t→(PSE e)→◯t
eval2S (Lit i p) env = castTa (sym p) ⟨i⟩
eval2S (V v) env = evalV2S v env
eval2S (App e1 e2) env = ⟨ ~(eval2S e1 env) ~(eval2S e2 env)⟩
eval2S (EInl e p) env =  castTa (sym p) ⟨Left ~(eval2S e env)⟩
eval2S (EInr e p) env =  castTa (sym p) ⟨Right ~(eval2S e env)⟩
eval2S (Abs pat body p) env = castTa (sym p)
                                ⟨\x→~(evalPat2S pat ⟨x⟩ env h)⟩
 where h (Nothing) = ⟨error "fail"⟩
       h (Just e) = eval2S body e
eval2S (ECase e matches) env =
  ⟨ let value = $(eval2S e env) in ~(evalCase2S ⟨value⟩ matches env)⟩

evalCase2S :: ◯t1 →[Match e t1 t2]→PSE e→◯t2
evalCase2S val [] env = ⟨error "fail"⟩
evalCase2S val ((Match (pat,body)):rest) env = evalPat2S pat val env h
   where h (Nothing) = evalCase2S val rest env
         h (Just env2) = eval2S body env2

evalVar2S :: Var e t→(PSE e)→◯t
evalVar2S (Z p) (EXT _ b p2) = castTa prf b
  where (_,prf) = pairParts (trans (sym p2) p)
evalVar2S (S s p) (EXT e _ p2) = evalVar2S s (castTa prf e)
  where (prf,_) =  (pairParts (trans (sym p2) p))
evalVar2S (Z p) (INIT env) = ⟨snd ~(castTa p env)⟩
evalVar2S (S s p) (INIT env) = evalVar2S s (INIT ⟨fst ~(castTa p env)⟩)

evalPat2S :: Pat t ein eout→◯t→(PSE ein)→
             (Maybe (PSE eout)→◯ans)→◯ans
evalPat2S (PVar  p) v ein k =  k (Just  (castTa (sym p) (ext ein v)))
evalPat2S (PInl pt p) v ein k =
  ⟨ case ~(castTa p v) of Left x  →~(evalPat2S pt ⟨x⟩ ein k)
                          Right x→~(k Nothing) ⟩
evalPat2S (PInr pt p) v ein k =
  ⟨ case ~(castTa p v) of Left x→~(k Nothing)
                          Right x→~(evalPat2S pt ⟨x⟩ ein k)⟩
evalPat2S (PPair pt1 pt2 p) v ein k =
  ⟨case ~(castTa p v) of
     (v1,v2)→~(evalPat2S pt1 ⟨v1⟩ ein (h ⟨v2⟩))⟩
 where h n Nothing = k Nothing
       h n (Just eout1) = evalPat2S pt2 n eout1 k
```

Figure 5.7: Staged interpreter for L_1^+ with binding time improvements.

Part III

Omega and Further Applications

Chapter 6

A Meta-language with Built-in Type Equality

[1]

6.1 Introduction

Earlier in this dissertation we looked at different ways of providing support for open heterogeneous meta-programming. First, we used a custom-designed meta-language with dependent types. Next we devised a methodology for supporting open heterogeneous meta-programming in Haskell.

However, practical experience with open heterogeneous meta-programming in Haskell does have one practical draw-back: it is tedious, requiring a lot of human intervention for rather simple tasks such as equality combinator manipulation. Since the combinator manipulation is pretty straightforward, albeit tedious, we became interested in extending the type system of Haskell to automate the manipulation of equality proofs as much as possible.

At this point we read Cheney and Hinze's work on phantom types [19]. Cheny and Hinze devise a type system that automatically propagates equalities between types, and solves type equality congruences. With this type system, we could easily implement all our examples in a much simpler, cleaner notation. Furthermore, Cheney and Hinze presented a proof that such a type system is type safe, and that type-checking is decidable. Finally, using this type system we no longer had to resort to axioms for manipulating equality types (e.g., `pairParts :: Equal (t1,t2) (t3,t4)` → `(Equal t1 t3, Equal t2 t4)`), which could not be implemented in Haskell itself, but had to be given as primitives.

Inspired by their idea we proceeded to experiment and design a functional programming language, based on Haskell, that implements their proposals along with some other features our experimentation in the previous chapter found might be useful. We called this language Omega [124]. Omega has proved to be a very useful vehicle for heterogeneous meta-programming, and much of its design was directly motivated

[1] Material from this chapter was published as [123] and [99].

by the kind of meta-programming we have demonstrated in this dissertation. We will familiarize the reader with Omega through the small tutorial offered in this chapter.

6.2 Omega: A Meta-language Supporting Type Equality

The essential characteristic of programming with type equality is the manipulation of the proofs of equalities between types using equality combinators. It has two practical drawbacks. First, manipulation of proofs using combinators is tedious. Second, while present throughout a program, the equality proof manipulations have no real computational content – they are used solely to leverage the power of the Haskell type system to accept certain programs that are not typable when written without the proofs. With all the clutter induced by proof manipulation, it is sometimes difficult to discern the difference between the truly important algorithmic part of the program and mere equality proof manipulation. This, in turn, makes programs brittle and rather difficult to change.

What if we could extend the type system of Haskell, in a relatively minor way, to allow the type-checker itself to manipulate and propagate equality proofs? That is the idea behind Omega [124]. In the remainder of this Chapter, we shall use Omega, rather than pure Haskell to write our examples. We conjecture that, in principle, whatever is possible to do in Omega, it is also possible to do in Haskell (plus the usual set of extensions), only in Omega it is expressed more cleanly and succinctly.

The syntax and type-system of Omega was designed to closely resemble Haskell (with GHC extensions). For practical purposes, we could consider (and use) it as a conservative extension to Haskell. In this section, we will briefly outline only the relevant differences between Omega and Haskell.

6.3 An Omega Primer

Omega is implemented as a stand-alone interpreted language, similar to the Hugs implementation of Haskell. Using a rudimentary module system, the user can load, type-check and execute source files that closely resemble Haskell. In this section, we shall explain some essential features of Omega, informally and by example. The language Omega has many interesting features such as built-in type equality, the polymorphic and extensible kind system, support for staging. These features were motivated by the examples appearing in earlier chapters.

6.3.1 Data-types with Equality

Here, we shall discuss the most important difference between Haskell and Omega: the data-type definition. First, recall the definition of a type of well-typed λ-terms from Chapter 5.

```
data Exp e t
  = Lit Int (Equal t Int)
  | V (Var e t)
  | ∀αβ. Abs (Rep α) (Exp (e,α) β) (Equal t (α→β))
  | ∀α.  App (Exp e (α→t)) (Exp e α)

data Var e t
 = ∀γ.  Z (Equal e (γ,t))
 | ∀γα. S (Var γ t) (Equal e (γ,α))
```

These data-types rely on the data-type `(Equal a b)`, which is the type of proofs that the types `a` and `b` are equal. When constructing `Exp` or `Var` values, the user must construct and supply the required equality proofs.

In Omega, the equality between types is not encoded explicitly (using the type constructor `Equal`), but, rather, it is built-in and implicit. Let us reformulate the well-typed λ-terms using Omega syntax:

```
data Exp e t
  = Lit Int where  t=Int
  | V (Var e t)
  | ∀αβ. Abs (Rep α) (Exp (e,α) β) where  t = (α→β)
  | ∀α.  App (Exp e (α→t)) (Exp e α)

data Var e t
 = ∀γ.  Z where e = (γ,t)
 | ∀γα. S (Var γ t) where e = (γ,α)
```

Each data-constructor in Omega may contain a `where` clause which contains a list of equations between types in scope of the definition. These equations play the same role as the `Equal` in our Haskell examples, with one important difference. The user is not required to provide any actual evidence of type equality – the Omega type checker keeps track of equalities between types and proves and propagates them automatically.

Cheney and Hinze formally define a type system with equality types [19]. We will quickly sketch out such a type system here, omitting most of the details. Figure 6.1 summarizes the Cheney and Hinze's typing judgments: a standard λ-calculus typing relation is augmented with equality contexts Ψ, which keep track of known equalities between types. An additional judgment, $\Delta; \Psi \vdash \tau_1 = \tau_2$, is defined to prove equalities

$$
\begin{array}{llll}
\sigma, \tau & ::= & \cdots & \text{Types} \\
\epsilon & ::= & \tau_1 = \tau_2 & \text{Type equations} \\
\Sigma & ::= & \cdot \mid \Sigma; \mathsf{data} T\ \overline{\alpha : \kappa} = \overline{\exists \overline{\beta : \kappa}. C\ \overline{\sigma}\ \mathsf{with}\ \overline{\epsilon}} & \text{Data-type signatures} \\
\Psi & ::= & \cdot \mid \Psi, \epsilon & \text{Equation contexts} \\
e & ::= & C\ [\overline{\tau}]\ \overline{e} \mid \mathsf{case}[\tau]\ e\ \mathsf{of}\ ms \mid \cdots & \text{Expressions: constructors and case} \\
ms & ::= & (C\ [\beta]\ \overline{x}\ \rightarrow\ e \mid ms) \mid \cdot & \text{Pattern matches}
\end{array}
$$

$$
\begin{array}{ll}
\Delta; \Psi \vdash \tau = \tau & \text{Type equivalence} \\
\Delta; \Psi; \Gamma \vdash e : \tau & \text{Typing expressions} \\
\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma & \text{Pattern match typing}
\end{array}
$$

$$
\boxed{\mathsf{data}\ T\ \overline{\alpha : \kappa} = \exists \overline{\beta : \kappa}. C\ \overline{\sigma}\ \mathsf{with}\ \overline{\epsilon} \in \Sigma}
$$

$$
\frac{\Delta; \Psi; \Gamma \vdash e_i : (\tau_i[\overline{\tau'}/\overline{\alpha}, \overline{\tau}/\overline{\beta}]) \quad \Delta; \Psi \vdash \epsilon_i[\overline{\tau'}/\overline{\alpha}, \overline{\tau}/\overline{\beta}]}{\Delta; \Psi; \Gamma \vdash C[\overline{\tau}]\overline{e} : T\ \overline{\tau'}}(\text{Cons}) \qquad \frac{\Delta; \Psi; \Gamma \vdash e : T\ \overline{\tau} \quad \Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma}{\Delta; \Psi; \Gamma \vdash \mathsf{case}[\tau]\ e\ \mathsf{of}\ ms : \sigma}(\text{Case})
$$

$$
\frac{\Delta; \Psi; \Gamma \vdash ms : T\,\overline{\tau} \Rightarrow \sigma \quad \Delta, \overline{\beta}; \Psi, \overline{\epsilon}[\overline{\tau}/\overline{\alpha}, \overline{\gamma}/\overline{\beta}]; \Gamma, \overline{x_n} : \sigma_n[\overline{\tau}/\overline{\alpha}, \overline{\gamma}/\overline{\beta}] \vdash e : \sigma}{\Delta; \Psi; \Gamma \vdash (C\ [\overline{\gamma}]\ \overline{x}\ \rightarrow e \mid ms) : T\,\overline{\tau} \Rightarrow \sigma}(\text{Match})
$$

$$
\frac{\Delta; \Psi; \Gamma \vdash e : \tau_1 \quad \Delta; \Psi \vdash \tau_1 = \tau_2}{\Delta; \Psi; \Gamma \vdash e : \tau_2}(\text{EqCoerce})
$$

Figure 6.1: Type system for Omega-like language (based on Cheney and Hinze).

between types. Data-types are defined as in the Omega examples above: each constructor definition may contain a set of equalities between types.

Novel typing rules for constructor application and case expressions are formulated in the following way:

1. When applying a constructor C, which is defined to require equations ϵ, those equations must be proven (using the equality judgment $\Delta; \Psi \vdash \tau_1 = \tau_2$) to hold based on the current equality context (rule Cons, Figure 6.1).

2. When taking apart a constructor value using case, an appropriate instantiation of the equations ϵ from the definition of the constructor are added to the equality context when type-checking the body of each case match (rules Case and Match, Figure 6.1).

3. Finally, a conversion rule that allows us to assign the type τ_2 to an expression that has the type τ_1, provided that we can prove that τ_1 equals τ_2 in the current equality context (rule EqCoerce, Figure 6.1).

For further details, the reader is referred to the Cheney and Hinze paper [19]. The Omega interpreter includes a a type checker for a similar type system, supporting many Haskell-like type system features and type inference. We briefly explain how such a type checker works in practice.

The mechanism Omega uses to keep track of equalities between types is very similar to the constraints that the Haskell type checker uses to resolve class-based overloading. A special qualified type [65] is used to assert equality between types, and a constraint solving system is used to simplify and discharge these assertions. When assigning a type to a type constructor, the equations specified in the where clause just become predicates in a qualified type. Thus, the constructor `Lit` is given the type `∀e t.(t=Int) => Int → Exp e t`. The equation `t=Int` is just another form of predicate, similar to the class membership predicate in the Haskell type (for example, `Eq a => a -> a -> Bool`).

When type-checking an expression, the Omega type checker keeps two sets of equality constraints.

Obligations. The first set of constraints is a set of *obligations.* For example, consider type-checking the expression `(Lit 5)`. The constructor **Lit** has the type `∀e t.(t=Int) => Int → Exp e t`. Since `Lit` is polymorphic in `e` and `t`, the type variable `t` can be instantiated to `Int`. Instantiating `t` to `Int` also creates the equality constraint obligation `Int=Int`, which can be trivially discharged by the type checker.

```
Lit 5 :: Exp e Int  with obligation Int = Int
```

One practical thing to note is that the data-constructors of `Exp` and `Var` are now given the following types:

```
Lit :: ∀e t.t=Int => Exp e t
V   :: ∀e t. Var e t → Exp e t
Abs :: ∀t t1 t2 e. t=(t1->t2) => Exp (e,t1) t2 → Exp e t
App :: ∀ e t1 t. Exp e (t1 → t) → Exp e t1 → Exp e t
```

It is important to note that the above qualified types can be *instantiated* to the same types that the smart constructors for well-typed abstract syntax have in Haskell. We have already seen this for `Lit`. Consider the case for `Abs`. First, the type variable `t` can be instantiated to `(t1 → t2)`. Now, the proof obligation introduced by the constructor is `(t1 → t2)=(t1 → t2)`, which can be immediately discharged. This leaves the type `Exp (e,t1) t2 → Exp e (t1 → t2)`.

Assumptions. The second set of constraints is a set of *assumptions* or *facts*. Whenever, a constructor with a `where` clause is pattern-matched, the type equalities in the where-clause are added to the current set of assumptions in the scope of the pattern. These assumptions can be used to discharge obligations. For example, consider the following partial definition:

```
evalList ::  Exp e t → e → [t]
evalList exp env =
 case exp of Lit n → [n]
```

When the expression exp of type `(Exp e t)` is matched against the pattern `(Lit n)`, the equality `t=Int` from the definition of `Lit` is introduced as an assumption.

The type signature of `evalList` induces the requirement that the right-hand side of the `case` expression have the type `[t]`. However, the right-hand side of the `case` expression, `[n]`, has the type `[Int]`. The type checker now must discharge (prove) the obligation `[t]=[Int]`, while using the fact, introduced by the pattern `(Lit n)` that `t=Int`. The Omega type-checker uses an algorithm based on congruence-closure [88], to discharge equality obligations.

In Haskell, the proof of this obligation would fall on the programmer, by explicitly constructing a proof value of type `(Equal [t] [Int])`, or using the function `castTa :: Equal a b -> f a -> f b` to cast from `[Int] to [t]`.

```
evalList :: Exp e t→e→[t]
evalList exp env =
  case exp of Lit n tInt→ castTa tInt [n]
```

In Omega, these proofs are constructed automatically, and this is perhaps the greatest practical benefit of Omega.

Another interesting example of programming in Omega is to re-implement, explicitly, the equality type `(Equal a b)`. Consider the following definition:

```
data Equal a b = Eq where a = b
```

Note that the constructor `Eq` requires no arguments. The type Omega assigns to it is `a=b => Equal a b`, which can be simplified to `Equal a a` – the same type as the Haskell equality combinator `self :: Equal a a`.

Since Omega's type system already knows how to manipulate equalities, writing equality proof combinators becomes trivial. Consider the transitivity combinator:

```
trans :: Equal a b→Equal b c→Equal a c
trans (ab@Eq) (bc@Eq) = Eq
```

First, matching the pattern `ab@Eq` introduces the assumption `a=b`. Similarly, the pattern `bc@Eq` introduces the assumption `b=c`. The result, `Eq` requires the proof obligation `a=c` to be discharged in order to return a value of type `Equal a c`. The congruence closure algorithm in the Omega type checker can then easily discharge this obligation based on the available assumptions.

Finally, we emphasize that, even though the examples in the chapters that follow are presented in Omega, they can all be implemented in Haskell as well, with the already alluded-to caveats that primitive equality

proof combinators such as `pairParts` may need to be used on the Haskell side. For the rest of this chapter (and subsequent chapters that use Omega), Omega should be considered as notational convenience. We have designed Omega to achieve a greater conciseness and clarity of presentation, because the explicit equality proof manipulation in meta-programs manipulating more complex object-languages can become very tedious.

6.3.2 Inductive Kinds

Let us recall the Haskell encoding of the natural numbers at the type level from Chapter 4. At the type level, natural number 0 is represented by the type `Z`, number 1 by the type `(S Z)`, 2 by `(S (S Z))`, and so on.

```
data Z =
data S x =
data IsNat n =      IsZero (Equal n Z)
            | ∀ m. IsSucc (IsNat m) (Equal n (S m))
```

This definition follows a standard pattern. First, each constructor of natural numbers is defined as a *type constructor*, `Z :: *` and `S :: * → *`, respectively. It is worth noting that there are no *values* classified by `Z`, `(S Z)`, and so on. This can be seen by the lack of constructor functions for `Z` and `S`. It is also worth noting that the type system of Haskell has no way of statically preventing the type constructors `Z` and `S` from being combined with other types to construct ill-formed representations that do not correspond to any natural number, such as `(S (S (Int → Bool)))`.

Second, we define a type constructor of *runtime representations* of natural numbers `IsNat :: * → *`. This data-type allows us to construct values that are classified by `IsNat` which are parameterized by *well-formed* natural numbers at the type level. In other words, `IsNat` reflects the natural numbers at the type level (comprised of `S` and `Z`) to the value level. The type constructor `IsNat` comes with a built-in invariant: for any value classified by the type `(IsNat n)`, the type `n` is a well-formed representation of some natural number.

The type constructor `IsNat` performs the role of a singleton type: there is only one valid value of type `IsNat n`, i.e., that which is isomorphic to the natural number `n`.[2] Type constructors such as `IsNat` allow the programmer to connect the type-level representations of naturals with the behavior of programs. For example, a function of the type `((IsNat m) → (IsNat (S (S m))))` takes any natural number as its argument and returns a natural number that is greater by 2.

[2]Note that there is no way in Haskell or Omega to check that a particular type constructor such as `IsNat` is indeed a singleton type. Rather, being a singleton is a meta-theoretical property that the programmer must maintain in writing his program.

value		type		kind		sort		···
5	::	Int	::	*0	::	*1	::	···
				Nat	::	*1	::	···
		Z	::	Nat	::	*1	::	···
		S	::	Nat ⇝ Nat	::	*1	::	···
		IsNat	::	Nat ⇝ *0	::	*1	::	···
IsZero	::	IsNat Z	::	*0	::	*1	::	···
IsSucc	::	(IsNat m) → (IsNat n)	::	*0	::	*1	::	···

Table 6.1: Classification in Omega.

In Omega, there is a simple feature that makes the encoding technique described above both simpler and more user friendly. This feature allows the programmer to define new *kinds*.

Before we demonstrate how kind declarations work, we shall explain the classification system of Omega. In Haskell, values and expressions are *classified* by types. In Omega, the classification scheme is somewhat more general. Values and expressions are classified by types, as in Haskell. Types themselves are classified by the *kind* `*0`. Kinds (e.g., `*0`) are classified by `*1`, `*1` by `*2` and so on. Kinds can be combined using a kind arrow (⇝). Table 6.1 gives an example of the classification relation in Omega.

To represent natural numbers at the type level in Omega, we shall define a new *kind* `Nat`:

```
kind Nat = Z | S Nat
```

The kind `Nat` has two *type constructors*: (1) `Z` of kind `Nat`; (2) `S` of kind Nat⇝Nat. For example, `(S Z)` is a valid type of kind `Nat`. It is important to note, however, that `(S Z)` is *not* a type of kind `*0`. Now, we can define a type of runtime representations of natural numbers. It is a type constructor `IsNat :: Nat⇝*0`:

```
data IsNat (n::Nat) =     IsZero where n = Z
                    | ∀m. IsSucc (IsNat m)
                                 where n = (S m)
one :: IsNat (S Z)
one = IsSucc IsZero
```

It is important to notice that the two versions of the example above, Haskell and Omega, are equally expressive: using two different type constructors for successor and zero works equally well as the Omega's kind declaration. The advantage of using Omega is that certain kind errors can be caught earlier, since the kind definition facility provides an additional amount of type discipline at the kind level which is missing in Haskell. Thus, the user cannot even write a type `(S Bool)`, since that would result in a kind error. Also, it allows us to combine all the constructors that represent values at the type level (with the same kind) in

Expressions and types

$$\begin{array}{lll} \tau & \in \mathbb{T} ::= & \mathsf{b} \mid \tau_1 \to \tau_2 \\ \Gamma & \in \mathbb{G} ::= & \langle\rangle \mid \Gamma, \tau \\ e & \in \mathbb{E} ::= & \mathsf{Var}\ n \mid \lambda_\tau e \mid e_1\ e_2 \end{array}$$

$$\frac{}{\Gamma, \tau \vdash 0 : \tau}(\text{Base}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau' \vdash (n+1) : \tau}(\text{Weak}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma \vdash \mathsf{Var}\, n : \tau}(\text{Var})$$

$$\frac{\Gamma, \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda_{\tau_1}.e : \tau_1 \to \tau_2}(\text{Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}(\text{App})$$

Substitutions à la $\lambda\upsilon$ [9]

$$\sigma \quad \in \mathbb{S} ::= \quad e/ \quad \mid \quad \Uparrow(\sigma) \quad \mid \uparrow$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e/ : \Gamma, \tau}(\text{Slash}) \qquad \frac{}{\Gamma, \tau \vdash \uparrow : \Gamma}(\text{Shift}) \qquad \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma, \tau \vdash \Uparrow(\sigma) : \Gamma', \tau}(\text{Lift})$$

Figure 6.2: Simply typed λ-calculus with substitutions.

one single definition which makes it easier for the programmer to modify and maintain.

6.4 Omega Example: Substitution

To round off the introduction to Omega we present a slightly larger example. First, we shall define a language of simply typed λ-calculus judgments, and then implement a type-preserving substitution function on those terms.

This example demonstrates type-preserving *syntax-to-syntax* transformations between object-language programs. Substitution, which we shall develop in the remainder of this Chapter, is one such transformation. Furthermore, a correct implementation of substitution can be used to build more syntax-to-syntax transformations. At the end of this Chapter, we shall provide an implementation of big-step semantics that uses substitution.

The substitution operation we present preserves object-language typing. Unlike the interpreters we have presented previously, it not only analyzes object-language typing judgments, but also builds new judgments based on the result of that analysis.

6.4.1 The Simply Typed λ-calculus with Typed Substitutions

Figure 6.2 defines two sets of typed expressions. The first set of expressions, presented in the top half of Figure 6.2 is just the simply typed λ-calculus. The second set of expressions, presented in the bottom

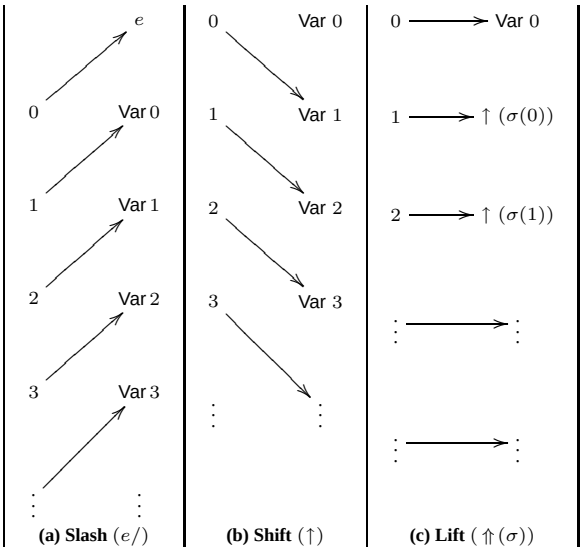


Figure 6.3: Substitutions

half of the figure defines a set of typed substitutions. The substitution expressions are taken from the $\lambda\upsilon$-calculus [9]. There are several of other ways to represent substitutions explicitly as terms (see Kristoffer Rose's excellent paper [113] for a comprehensive survey), but we have chosen the notation of $\lambda\upsilon$ for its simplicity.

A substitution expression σ is intended to represent a mapping from de-Bruijn indices to expressions (i.e., a substitution), the same way that λ-expressions are intended to represent functions. As in $\lambda\upsilon$, we define three kinds of substitutions in Figure 6.2 (see Figure 6.3 for a graphical illustration):

1. *Slash* $(e/)$. Intuitively, the slash substitution maps the variable with the index 0 to e, and any variable with the index $n+1$ to Var n.

2. *Shift* $(\uparrow)$. The shift substitution adjusts all the variable indices in a term by incrementing them by one. It maps each variable n to the term Var $(n+1)$.

3. *Lift* $(\Uparrow(\sigma))$. The lift substitution $(\Uparrow(\sigma))$ is used to mark the fact that the substitution σ is being applied to a term in a context in which index 0 is bound and should not be changed. Thus, it maps the variable with the index 0 to Var 0. For any other variable index $n+1$, it maps it to the term that σ maps to n, with the provision that the resulting term must be adjusted with a shift: $((n+1) \mapsto \uparrow(\sigma(n)))$.

Typing substitutions. The substitution expressions are typed. The typing judgments of substitutions, written $\Gamma_1 \vdash \sigma : \Gamma_2$, indicate that the type of a substitution, in a given type assignment, is another type

assignment. The intuition behind the substitution typing judgment is the following: the type assignment Γ_1 assigns types to the free variables that may occur in the expressions that are a part of the substitution σ; the type assignment Γ_2 assigns types to the free variables in any expression that the substitution σ may act upon.

Example. We describe a couple of example substitutions.

1. Consider the substitution $(\mathsf{True}/)$. This substitution maps the variable with the index 0 to the Boolean constant `True`. The type of this substitution is $\Gamma \vdash \mathsf{True}/ : \Gamma, \mathsf{Bool}$. In other words, given any type assignment, the substitution $(\mathsf{True}/)$ can be applied in any context where the variable 0 is assigned type Bool.

2. Consider the substitution $\sigma = (\Uparrow(\mathsf{True}/))$. σ is the substitution that replaces the variable with the index 1 with the constant True.

   Recall that the type of any substitution θ under a type assignment Γ, is a type assignment Δ (written $\Gamma \vdash \theta : \Delta$), such that for any expression e' to which the substitution θ is applied, the following must hold $\Delta \vdash e' : \tau$ and $\Gamma \vdash \theta(e') : \tau$.

   So, what type should we assign to σ? When applied to an expression, a lift substitution ($\sigma = \Uparrow(\mathsf{True}/)$) does not change the variable with the index 0. Thus, when typing σ as $\Gamma \vdash \sigma : \Delta$, we know something about the shape of Γ and Δ. Namely, for some Δ', we know that $\Delta = (\Delta', \tau)$, and for some Γ', we know that $\Gamma = (\Gamma', \tau)$. The type assignments Δ' and Γ' are determined by the sub-substitution $\mathsf{True}/$, yielding the following typing derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{True} : \mathsf{Bool}}\text{Const}}{\Gamma \vdash \mathsf{Bool}/ : \Gamma, \mathsf{Bool}}\text{Slash}}{\Gamma, \tau \vdash \Uparrow(\mathsf{Bool}/) : \Gamma, \mathsf{Bool}, \tau}\text{Lift}$$

We briefly explain the typing rules for the substitutions (Figure 6.2):

1. *Slash* $(e/)$. A slash substitution $e/$ replaces the 0-index variable in an expression by e. Thus, in any context Γ, where e can be given type τ, the typing rule requires the substitution to work only on expressions in the type assignment Γ, τ, where the 0-index variable is assigned the type τ.

$$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash e/ : \Gamma, \tau}(\text{Slash})$$

2. *Shift (*$\uparrow$*).* Since the shift substitution maps all variables with index n to a variable with index $n + 1$, this means that, whatever a type assignment assigned to the index 0, prior to the substitution, the

Substitution on expressions

$(\cdot,\cdot) \Rightarrow \cdot \subset \mathbb{S} \times \mathbb{E} \times \mathbb{E}$

$$\frac{(\sigma, e_1) \Rightarrow e_1' \quad (\sigma, e_2) \Rightarrow e_2'}{(\sigma, (e_1\ e_2)) \Rightarrow e_1'\ e_2'} \qquad \frac{(\Uparrow(\sigma), e) \Rightarrow e'}{(\sigma, \lambda.e) \Rightarrow \lambda e'} \qquad \frac{(\sigma, n) \Rightarrow e}{(\sigma, \mathsf{Var}\ n) \Rightarrow e}$$

Substitution on variables

$(\cdot,\cdot) \Rightarrow \cdot \subset \mathbb{S} \times \mathbb{N} \times \mathbb{E}$

$$\frac{}{(e/, 0) \Rightarrow e} \qquad \frac{}{(e/, n+1) \Rightarrow \mathsf{Var}\ n} \qquad \frac{}{(\Uparrow(\sigma), 0) \Rightarrow \mathsf{Var}\ 0}$$

$$\frac{(\sigma, n) \Rightarrow e \quad (\uparrow, e) \Rightarrow e'}{(\Uparrow(\sigma), n+1) \Rightarrow e'} \qquad \frac{}{(\uparrow, n) \Rightarrow \mathsf{Var}\ (n+1)}$$

Figure 6.4: Applying substitutions to terms

substitution can be contracted because after the substitution is performed 0-index variable no longer occurs.

$$\frac{}{\Gamma, \tau \vdash \uparrow : \Gamma}(\text{Shift})$$

3. *Lift* ($\Uparrow(\sigma)$). For any variable index $(n+1)$ in a term, the substitution $\Uparrow(\sigma)$ applies σ to n and then shifts the resulting term. Thus, the 0-index term in the type assignment remains untouched, and the rest of the type assignment is as specified by σ:

$$\frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma, \tau \vdash \Uparrow(\sigma) : \Gamma', \tau}(\text{Lift})$$

Applying substitutions. In the remainder of this Section, we show how to implement a function (we call it subst) that takes a substitution expression σ, a λ-expression e, and returns an expression such that all the indices in e have been replaced according the substitution. In the simply typed λ-calculus, substitution preserves typing, so we expect the following property to be true of the substitution function subst: if $\Gamma \vdash \sigma : \Delta$ and $\Delta \vdash e : \tau$, then $\Gamma \vdash \mathsf{subst}\ \sigma\ e : \tau$.

How should subst work? Figure 6.4 presents two judgments, $(\sigma, e_1) \Rightarrow e_2$ and $(\sigma, n) \Rightarrow e$, which describe the action of substitutions on expressions and variables, respectively. These judgments are derived from the reduction relations of the $\lambda\upsilon$-calculus [9]. It is not difficult to show that this reduction strategy indeed does implement capture avoiding substitution, although we omit such proof here to avoid unnecessary digression (see Benaissa, Lescanne & al. [9] for proofs).

Next, we show how to implement this substitution operation in Omega, using expression and substitution judgments instead of expressions and substitution expressions.

6.4.2 Judgments

The expression and substitution judgments can be easily encoded in Omega. The data-types `Var` and `Exp` encode expression and variable judgments presented in Figure 6.2. We have only added a constructor `Const` for constant expressions in order to be able to write more interesting examples. The λ-calculus fragment is identical to the one presented earlier in this chapter, and we shall not belabor its explanation.

```
data Var e t = ∀d.    Z where e = (d,t)
             | ∀d t2. S (Var d t) where e = (d,t2)

data Exp e t =          V   (Var e t)
             | ∀t1 t2. Abs (Exp (e,t1) t2) where t = t1→t2
             | ∀t1.    App (Exp e (t1→t)) (Exp e t1)
             |         Const t
```

Next, we define a data-constructor `Subst gamma delta` that represents the typing judgments for substitutions . The type constructor `Subst gamma delta` represents the typing judgment $\Gamma \vdash \sigma : \Delta$ presented in Figure 6.2.

```
data Subst gamma delta =
  ∀t1.    Shift where gamma = (delta,t1)
| ∀t1.    Slash (Exp gamma t1) where delta = (gamma,t1)
| ∀del1 gam1 t1. Lift (Subst gam1 del1)
                 where delta = (del1,t1), gamma = (gam1,t1)
```

6.4.3 Substitution

Finally, we define the substitution function `subst`. It has the following type:

```
subst :: Subst gamma delta -> Exp delta  t -> Exp gamma t
```

It takes a substitution whose type is `delta` in some type assignment `gamma`, an expression of type `t` that is typed in the type assignment `delta`, and produces an expression of type `t` typable in the type assignment `gamma`.

We will discuss the implementation of the function `subst` (Figure 6.5) in more detail. In several relevant cases, we shall describe the process by which the Omega type-checker makes sure that the definitions

```
1 subst :: Subst gamma delta→Exp delta t→Exp gamma t
2 subst s (App e1 e2)  = App (subst s e1) (subst s e2)
3 subst s (Abs e) = Abs (subst (Lift s) e)
4 subst (Slash e) (V Z) =  e
5 subst (Slash e) (V (S n)) = V n
6 subst (Lift s) (V Z) = V Z
7 subst (Lift s) (V (S n)) = subst Shift (subst s (V n))
8 subst  Shift (V n) = V (S n)
```

Figure 6.5: Substitution in simply typed λ-calculus.

are given correct types. Recall that every pattern-match over one of the `Exp` or `Subst` judgments may introduce zero or more equations between types, which are then available to the type-checker in the body of a case (or function definition). The type checker may use these equations to prove that two types are equal. In the text below, we sometimes use the type variables `gamma` and `delta` for notational convenience, but also Skolem constants like `_1134`. These are an artifact of the Omega type-checker (they appear when pattern-matching against values that may contain existentially quantified variables) and should be regarded as type constants.

1. The application case (line 2) simply applies the substitution to the two sub-expression judgments and then rebuilds the application judgment from the results.

2. The abstraction case (line 3) pushes the substitution under the λ-abstraction. It may be interesting to examine the types of the various subexpressions in this definition.

`Abs e`	:	`Exp delta t`, where `t=t1→t2`
`e`	:	`Exp (delta,t1) t2`
`s`	:	`Subst gamma delta`
`Lift s`	:	`Subst (gamma,t1) (delta,t1)`
`subst (Lift s) e`	:	`Exp (gamma,t1) t2`

   The body of the abstraction, `e` has the type `(delta,t1)`, where `t1` is the type of the domain of the λ-abstraction. In order to apply the substitution `s` to the body of the abstraction (`e`), we need a substitution of type `(Subst (gamma,t1) (delta,t1))`. This substitution can be obtained by applying `Lift` to `s`. Then, recursively applying `subst` with the lifted substitution to the body `e`, we obtain an expression of type `(Exp (gamma,t1) t2)`, from which we can construct a λ-abstraction of the `(Exp gamma (t1→t2))`.

3. The variable-slash case (line 4–5). There are two cases when applying the slash substitution to a variable expression:

   (a) Variable 0. The substitution `(Slash e)` has the type `(Subst (gamma) (gamma,t))`, and contains the expression `e :: Exp gamma t`. The expression `(V Z)` has the type `(Exp (delta,t) t)`. Pattern matching introduces the equation `gamma=delta`, and we can use `e` to replace `(V Z)`.

   ```
   | Slash e  ::  (Subst (gamma) (gamma,t))  |
   | e         ::  Exp gamma t                |
   ```

   (b) Variable $n + 1$. Pattern matching on the substitution argument introduces the equation `delta=(gamma,t1)`. Pattern matching against the expression `(V (S n))` introduces the equation `delta=(gamma',t)`, for some `gamma'`. The expression result expression `(V n)` has the type `(Exp gamma' t)`. The type checker then uses the two equalities to prove that it has the type `(Exp gamma t)`. It does this by first using congruence to prove that gamma=gamma', and then by applying this equality to obtain `Exp gamma' t = Exp gamma t`.

   ```
   | Slash e     ::  Subst gamma (gamma,t)  |
   | (V (S n))   ::  Exp delta t            |
   ```

4. The variable-lift case (lines 6–7). There are two cases when applying the lift substitution to a variable expression.

   (a) Variable 0. This case is easy because the lift substitution places makes no changes to the variable with the index 0. We are able simply to return `(V Z)` as a result.

   (b) Variable $n+1$. The first pattern (`Lift s :: Subst gamma delta`), on the substitution, introduces the following equations:

   ```
   delta = (d',_1),
   gamma = (g',_1)
   ```

   The pattern on the variable (`V(S n):: Var delta t`) introduces the equation

   ```
   delta = (d2,_2)
   ```

   The first step is to apply the substitution `s` of type `(Subst g' d')` to a decremented variable index `(V n)` which has the type `n :: Var d2 t`. To do this, the type checker has to show that `g'=d2`, which easily follows from the equations introduced by the pattern, yielding a result of type `(Exp g' t)`. Applying the `Shift` substitution to this result yields an expression of type `(Exp (g',a) t)` (where `a` is can be any type). Now, equations above

can be used to prove that this expression has the type `(Exp gamma t)` using the equation `gamma=(g',_1)`.

5. Variable-shift case (line 8). Pattern matching on the `Shift` substitution introduces the equation `gamma = (delta,_1)`. The expression has the type `(Exp delta t)`. Applying the successor to the variable results in an expression `(V (S n))` of type `(Exp (delta,a) t)`. Immediately, the type checker can use the equation introduced by the pattern to prove that this type is equal to `(Exp gamma t)`.

We have defined type-preserving substitution simply typed λ-calculus judgments. It is worth noting that Omega has proven very helpful in writing such complicated functions: explicitly manipulating equality proofs for such a function in Haskell, would result in code that is both extremely verbose and difficult to understand.

6.4.4 A Big-step Evaluator

Finally, we implement a simple evaluator based on the big-step semantics for the λ-calculus. The evaluation relation is given by the following judgment:

$$\frac{}{\lambda e \Rightarrow \lambda e} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow \lambda e' \quad (e_2/, e') \Rightarrow e_3 \quad e_3 \Rightarrow e''}{e_1\ e_2 \Rightarrow e''}$$

Note that in the application case, we first use the substitution $(e_2/, e') \Rightarrow e_3$ to substitute the argument e_2 for the variable with index 0 into the body of the λ-abstraction.

A big-step evaluator differs from the other interpreters for object languages we have presented in this dissertation. Whereas the other interpreters map object-language judgments to some related domain of values, the big-step evaluator is implemented as the function `eval` which takes a well-typed expression judgment of type `(Exp delta t)`, and returns judgments of the same type. The evaluator reduces β-redices using a call-by-name strategy, relying upon the substitution implemented above.

```
eval :: Exp delta t -> Exp delta t
eval (App e1 e2) =
  case eval e1 of
    Abs body -> eval (subst (Slash e2) body)
eval x = x
```

Note that the type of the function `eval` statically ensures that it preserves the typing of the object language expressions it evaluates, with the usual caveats that the `Exp`s faithfully encode well-typed λ-expressions.

Finally, let us apply the big-step evaluator to a simple example. Consider the expression, `example`.

```
example :: Exp gamma (a → a)
example = (Abs (V Z)) `App` ((Abs (Abs (V Z))) `App` (Abs (V Z)))
-- example = (λ x.x) ((λ y. (λ z.z))) (λ x.x)
```

The expression `example` evaluates the identity function. Applying `eval` to it yields precisely that result:

```
evExample = eval example
-- evExample = (Abs (V Z)) : Exp gamma (a → a)
```

Chapter 7

Example: $\lambda^{\Box}$

Up until now, we have considered object-languages based on the simply typed λ-calculus. In this section, we shall expand our range of object-languages by first providing implementations of well-typed interpreters for two object languages whose type systems are somewhat different from the type-system of the meta-language. These languages, we shall call them $L_{\Box}$ and $L_{\bigcirc}$, are based on the two extension of the typed λ-calculus, with modal and temporal operators, $\lambda_{\Box}$ [31] and $\lambda_{\bigcirc}$ [29].

Why these particular languages? First, they are interesting typed languages in their own right, as useful formalisms for describing two different kinds of staged computation. Second, formalizing their type systems in a Haskell-like language to obtain sets of well-typed object terms is a more challenging task, allowing us to showcase our heterogeneous meta-programming methodology.

The calculus $\lambda^{\Box}$ is an extension of the simply typed λ-calculus. This calculus was defined by Davies and Pfenning as the language of proof-terms for propositions in the necessity fragment of the intuitionistic modal logic S4 [30, 31]. The propositions in this logic (and, hence, types in $\lambda_{\Box}$) come equipped with the modal necessity (also called "necessitation") operator $\Box$.

Logically, the box operator expresses propositions that are *necessarily true* (the term *valid* is also used). For example, $\Box(a \to a)$, is such a proposition since $(a \to a)$ is always true, irrespective of the truth-value of a.

$\lambda_{\Box}$ is a (homogeneous) meta-programming language. The logical box operator used to classify types of *object*-programs (of $\lambda_{\Box}$). For example, the type $(\text{Int} \to \Box\text{String})$ in $\lambda_{\Box}$ is a type of a program generator that takes an integer and produces a piece of code that, when executed, yields a string value. Davies and Pfenning prove certain binding time separation properties [31] that guarantee that, for example, while the program of type $(\text{Int} \to \Box\text{String})$ generates the residual program of type String, all computation pertaining to its integer argument is performed while the residual program is being constructed, i.e., there is no leftover earlier stage computation in the residual program.

In this section, we shall present a small object language, called $L_{\Box}$ that is based on the type system of

b	$\in \mathbb{B} ::=$	$\mathsf{Int} \mid \cdots$	base types
τ	$\in \mathbb{T} ::=$	$\mathsf{b} \mid \tau \to \tau \mid \Box\tau \mid \tau \times \tau$	types
Γ, Δ	$\in \mathbb{G} ::=$	$\langle\rangle \mid \Gamma, \tau$	type assignments
e	$\in \mathbb{E} ::=$	$c \mid 0_{\mathsf{L}} \mid \uparrow e \mid \lambda\tau.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e$	λ-fragment
	$\mid$	$0_{\mathsf{R}} \mid e \uparrow \mid \mathsf{box}\ e \mid \mathsf{let\ box}\ e_1\ \mathsf{in}\ e_2$	modal fragment

Figure 7.1: The syntax of the language $L_{\Box}$.

$\lambda^{\Box}$.

7.1 Syntax of $L_{\Box}$.

The core syntax of the language $L_{\Box}$ is given in Figure 7.1. Types in $L_{\Box}$ are either base types such as Int, Bool, function types, box types or products. Expressions are somewhat non-standard and we need to explain them.

Variables. In standard formalizations of $\lambda^{\Box}$ [31], there are usually *two* (distinct) sets of variables. The fist is the set of variables bound by λ-abstractions. The second set is the set of modal variables that range over code fragments (box values), and are bound by the let box expressions. Following Davies and Pfenning, we shall call the former variables (0_{R}) we shall call *value variables*, and the latter (0_{L}) *modal variables*.

As is usual in examples we have presented so far, we opt for a de Bruijn style of variable naming. The name of each variable is a natural number indicating the number of intervening binding sites between the use and the definition of a variable. In Chapter 5, variables are represented by natural numbers. This has required us to formulate a separate auxiliary typing judgment for variables. Here, we slightly modify the notation for variables, following the example of Chen and Xi [18], who adopt their notation from the study of λ-calculus with explicit substitutions (See Kristoffer H. Rose's excellent tutorial [113] for more about explicit substitutions.)

In this notation, there is only one syntactic form for variables, corresponding to the index 0. Since in $L_{\Box}$ we have two separate sets of variables, we shall use two such expressions, 0_{L} for value variables, and 0_{R} for modal variables.

Variables at higher indices are obtained by a "shift" (e.g.,[71]) syntactic construct ($\uparrow e$ and $e \uparrow$, for value and modal variables, respectively, where $\uparrow$, on the left or on the right, binds more tightly than application). Intuitively, the expression $e \uparrow$ increments the indices of all free value variables in e by one.

We find this notation slightly more concise in practice, and include it here to simplify our presentation, since it allows us to, among other things, write only one `eval` function, dispensing with the auxiliary

function `lookUp` of Chapter 5.

In Figure 7.2 we give a few examples of programs in a $\lambda^{\Box}$-based programming language with named variables, and their equivalent in the formalism of $L_{\Box}$.

$\lambda^{\Box}$ term	$L_{\Box}$ term
$\lambda x.\lambda y.(x, y)$	$\lambda.\lambda.(\uparrow 0_{\mathsf{L}}, 0_{\mathsf{L}})$
let box $u = \mathsf{box}(1+2)$ in let box $v = \mathsf{box}(3+4)$ in box (v, u)	let box $(\mathsf{box}\ (1+2))$ in let box $(\mathsf{box}\ (3+4))$ in box$(0_{\mathsf{R}}, 0_{\mathsf{R}}\uparrow)$
power :: Int $\to \Box(\mathsf{Int} \to \mathsf{Int})$ power $0 = \mathsf{box}(\lambda x : \mathsf{Int}.\, 1)$ power $(n+1) =$ let box u = power n in box$(\lambda x.\, x * (u\ x))$	power = fix powerF powerF $= \lambda\mathsf{Int} \to \Box(\mathsf{Int} \to \mathsf{Int}).\ \lambda\mathsf{Int}.$ if $(0_{\mathsf{R}} == 0)$ then box $(\lambda\mathsf{Nat}.1)$ else let box$((0_{\mathsf{R}}\uparrow)\ (0_{\mathsf{R}} - 1))$ in box$(\lambda\mathsf{Int}.0_{\mathsf{R}} * (0_{\mathsf{L}}\ 0_{\mathsf{R}}))$

Figure 7.2: A comparison between $\lambda^{\Box}$ and $L_{\Box}$ syntax.

Box and Unbox. The two novel expression forms in $L_{\Box}$ are box and let box, which act as introduction and elimination forms for the box types. The expression (box e) acts as a form of quasi-quotation. It constructs an object-language program e. The expression (let box e_1 in e_2) takes an object-language program e_1, runs it, and binds its value to a box variable 0_{L} in the body of the expression e_1.

Products and constants. In defining $L_{\Box}$ we shall also assume that we have a number of other, uncontroversial simple types such as products. Furthermore, we will assume that for various base types such as integers, booleans and so on, we have a sufficient number of constants (including operations like addition, comparison, and so on) for practical purposes. We will show later how such constants can be elegantly embedded into Haskell encodings of $L_{\Box}$ typing judgments.

7.2 Type System of $L_{\Box}$

The type system of $L_{\Box}$ is given by the typing judgment relation ($(\Delta; \Gamma \vdash e : t) \subseteq \mathbb{G} \times \mathbb{G} \times \mathbb{E} \times \mathbb{T}$) in Figure 7.3. The first thing to notice is that there are two type assignments, Δ and Γ. The intuition behind this is that the λ-fragment of $L_{\Box}$ is typed in the usual fashion using the type assignment Γ. Since $\Box$ represents closed code, boxed expressions can be well-typed only when Γ is the empty type assignment (see rule Box in Figure 7.3). However, variables that range over code fragments can still be used inside boxed expression, and their types are recorded by the type assignment Δ. This allows us type-check expressions

that combine smaller box fragments into larger ones.

The λ fragment

$$\frac{}{\Delta;\Gamma,\tau \vdash 0_{\mathsf{R}} : \tau}\text{R-Var} \quad \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma,\tau' \vdash e \uparrow : \tau}\text{R-Shift}$$

$$\frac{\Delta;\Gamma,\tau_1 \vdash e : \tau_2}{\Delta,\Gamma \vdash \lambda\tau_1.e : \tau_1 \to \tau_2}\text{Abs} \quad \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau_2}\text{App}$$

The modal fragment

$$\frac{}{\Delta,\tau;\Gamma \vdash 0_{\mathsf{L}} : \tau}\text{L-Var} \quad \frac{\Delta;\Gamma \vdash e : \tau}{\Delta,\tau';\Gamma \vdash \uparrow e : \tau}\text{L-Shift}$$

$$\frac{\Delta;\langle\rangle \vdash e : \tau}{\Delta;\Gamma \vdash \mathsf{box}\ e : \Box\tau}\text{Box} \quad \frac{\Delta;\Gamma \vdash e_1 : \Box\tau_1 \quad \Delta,\tau_1;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{let\ box}\ e_1\ \mathsf{in}\ e_2 : \tau}\text{Unbox}$$

Figure 7.3: The Static Semantics of $L_{\Box}$

The λ-fragment. The rules for the modal fragment are the standard rules for the simply type λ-calculus, except where it comes to the treatment of variables. A variable expression 0_{L} implements the start rule of looking up its type from the rightmost position in the type assignment Γ. The rule for shift (L-Shift) implements the weakening – the expression e is typed in a smaller type assignment. The rules for abstraction, applications, and products (not shown in the figure) are completely standard.

The modal fragment. The static semantics of the modal fragment consists of four typing rules in Figure 7.3:

- *L-Var.* and *L-Shift* are the lookup and weakening cases for the set of non-lambda-bound variables. They are the same as their λ fragment counterparts, except that they use the type assignment Δ.
- *Box*. The box takes a sub-expression e, and type-checks it in the empty λ-fragment type assignment. If, under that assumption, the expression e has type τ, then the whole expression $\mathsf{box}\ e$ has the type $\Box\tau$.

  This captures the modal inference rule about necessity: e is a proof of a necessarily true proposition τ only if τ can be proven with no assumptions (indicated by the empty type assignment). Note, that while type-checking e we *are* allowed to use any variables that are typed in the type assignment Δ, since the type assignment Δ, as we will see, is augmented only with types that are themselves necessarily true.

- *Unbox.* The let box expression is an elimination construct for boxed expression. It takes two subexpressions, e_1 and e_2. Then, the expression e_1 must be shown to have some type $\Box\tau_1$. If this is the case, we are allowed to introduce an additional assumption (bind a variable) in the type assignment Δ that has the type τ_1. If, with such an augmented Δ we can prove that the expression e_2 has type τ, then we may conclude that the entire expression has the type τ.

  Note that this is the only rule in which the modal type assignment Δ is extended. Moreover, it is extended only with a type of a closed code fragment. Intuitively, the modal variables can occur free inside boxed expression precisely because we know that they only range over expressions that are themselves closed.

Examples. Finally, we give a couple of examples involving expressions with box types.

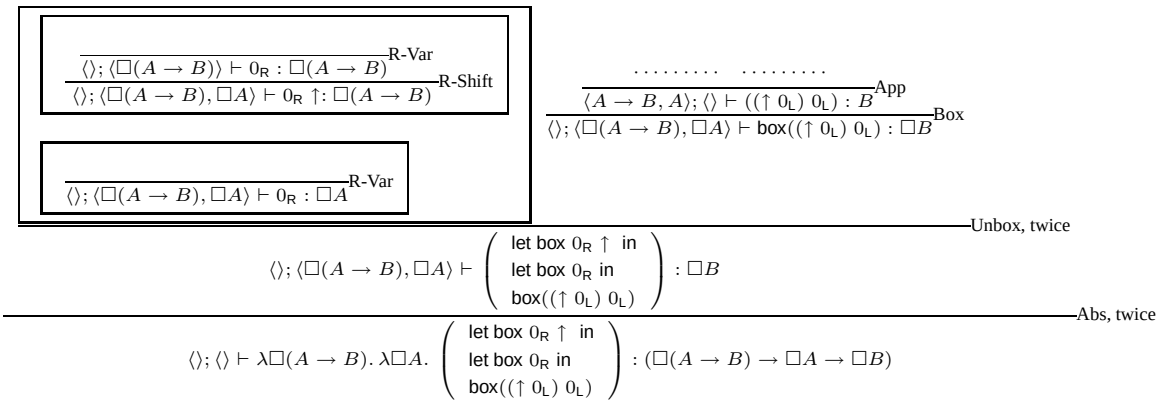
Consider the expression example1 (for some type A):

$$\begin{aligned}&\mathsf{example1} : \Box A \to A\\ &\mathsf{example1} = \lambda\Box A.\ \mathsf{let\ box}\ 0_{\mathsf{R}}\ \mathsf{in}\ 0_{\mathsf{L}}\end{aligned}$$

The type of this expression tells us, in terms of logic, that if A is necessarily true, than A is true. The typing derivation is listed below:

$$\dfrac{\dfrac{}{\langle\rangle;\langle\rangle,\Box A \vdash 0_{\mathsf{R}} : \Box A}\text{R-Var} \quad \dfrac{}{\langle\rangle, A;\langle\rangle,\Box A \vdash 0_{\mathsf{L}} : A}\text{L-Var}}{\langle\rangle;\langle\rangle,\Box A \vdash \mathsf{let\ box}\ 0_{\mathsf{R}}\ \mathsf{in}\ 0_{\mathsf{L}} : A}\text{Unbox}}{\langle\rangle;\langle\rangle \vdash \lambda\Box A.\mathsf{let\ box}\ 0_{\mathsf{R}}\ \mathsf{in}\ 0_{\mathsf{L}} : (\Box A \to A)}\text{Abs}$$

$$\begin{aligned}&\mathsf{example2} : \Box(A \to B) \to \Box A \to \Box B\\ &\mathsf{example2} = \lambda\Box(A \to B).\ \lambda\Box A.\ \begin{pmatrix}\mathsf{let\ box}\ 0_{\mathsf{R}} \uparrow\ \mathsf{in}\\ \mathsf{let\ box}\ 0_{\mathsf{R}}\ \mathsf{in}\\ \mathsf{box}((\uparrow 0_{\mathsf{L}})\ 0_{\mathsf{L}})\end{pmatrix}\end{aligned}$$



7.3 Encoding $L_{\Box}$ in Omega

A first step is to encode the $L_{\Box}$ judgments described in Figure 7.3 into an Omega(or, with slight modifications, Haskell) data-type. We shall use the technique that should be familiar to the reader by how of representing the typing judgment $\Delta; \Gamma \vdash e : \tau$ by a Haskell type constructor `Exp delta gamma tau`. Note that with Omega, there is no need to implement a set of *smart constructors* for the data-type defined in Figure 7.4.

```
data Exp Δ Γ t =
  ∀ Γ'.       VarR                                  where Γ = (Γ',t)
| ∀ Γ' t'.    ShiftR   (Exp Δ Γ' t)                 where Γ=(Γ',t')
| ∀ t1 t2.    Abs      (Exp Δ (Γ,t1) t2)            where t=(t1 → t2)
| ∀ t1.       App      (Exp Δ Γ (t1 → t)) (Exp Δ Γ t1)

|             Lift     t String

| ∀ Δ'.       VarL                                  where Δ=(Δ',t)
| ∀ Δ' t'.    ShiftL   (Exp Δ' Γ t)                 where Δ=(Δ',t')
| ∀ t1.       BoxExp   (Exp Δ () t1)                where t=(Box t1)
| ∀ t1.       UnBox    (Exp Δ Γ (Box t1)) (Exp (Δ,t1) Γ t)
```

Figure 7.4: Typing judgments of $L_{\Box}$ in Haskell.

The type assignments are represented by a nested product type. The lambda-calculus fragment is completely standard, as in L_0 (Chapter 5).

The judgment described in Figure 7.4 also contains the constructor `(Lift t String ::`

`t → Exp a b t)`. This constructor represents constants in the object language. It can be used to inject any Omega value (of type `t`) into `Exp`. It also takes a string argument that represents the name of the constant, for pretty-printing purposes. For example, the constant plus is encoded by simply lifting the addition operator: `(Lift (+) "+") :: Exp d g (Int → Int → Int)`.

Next, we consider the encoding of the modal fragment. The constructor `BoxExp` is used to create judgments of boxed terms. It has one argument, a judgment of type `Exp Δ () t1`. This ensures that the boxed expression is closed – any mention of free value variables will require the value type assignment to be a pair, causing a type mismatch with the requirement that the body expression have the type `()`. For example, the judgment for the $L_\Box$ term $(\mathsf{box}\ (\lambda.\, 0_\mathsf{R}))$ is represented by the Omega declaration `example1`, given below:

```
example1 :: Exp a b (Box (c -> c))
example1 = Box (Abs VarR)
```

However, if we try to create the judgment for the term $(\mathsf{box}\ 0_\mathsf{R})$, which cannot be correctly typed, the Omega type-checker complains with the following error message:

```
Lambdabox> box varr
ERROR - Type error in application
*** Expression     : box varr
*** Term           : varr
*** Type           : Exp c (d,b) b
*** Does not match : Exp a () b
```

The `where`-clause in the definition of the constructor specifies a proof obligation that `t` is equal to `Box t1`. The type constructor `Box` here is some, as yet undefined representation of boxed values. We will consider how to define Box later on.

Example: Power function. Here we shall construct an example $L_\Box$ well-typed program. The function `power` from In Figure 7.2 we show an integer exponentiation function `power`. This function can be staged based on the situation where its exponent argument is known. Thus, in $L_\Box$, power is given the type `(Int → □(Int → Int))`: given an integer exponent argument n, `power` generates a residual program that computes x^n, given its argument x.

Figure 7.5 shows the definition of the `power` function in the Omega encoding of $L_\Box$. We examine this definition more closely:

- In the first line, we can see that `power` is defined by using recursion. The $L_\Box$ constant `fixpoint`

```
power  :: Exp a b (Integer → Box (Integer → Integer))
power  = fixpoint ‘App‘ (Abs $ Abs $ Body)
           where body     = iffun ‘App‘ cond ‘App‘ zerocase ‘App‘ ncase
                 cond     = eq ‘App‘ varr ‘App‘ (Lift 0 ”0”)
                 zerocase = Box (Abs (int 1))
                 ncase    = Unbox reccall newbox
                 reccall  = (ShiftR VarR) ‘App‘ (minusone ‘App‘ VarR)
                 newbox   = Box (Abs $ times ‘App‘ VarR ‘App‘ (VarL ‘App‘ VarR))
                 minusone = Lift (\x → x-1) ”dec”
                 times    = Lift (\x y → x * y) ”times”
                 iffun    = Lift (\x y z → if x then y else z) ”if”
                 eq       = Lift (==) ”==”

fixpoint :: Exp d g ((a → a) → a)
fixpoint = Lift fix ”fix”
           where fix f    = f (fix f)
```

Figure 7.5: The staged `power` function in $L_{\Box}$.

is applied to a functional `(abs $ abs $ body)`, where the first abstracted variable represents the recursive call to the function `power`, and the second argument is the exponent n.

- The body of function `power` is a conditional expression that compares the exponent to 0, and then takes two cases:

  1. `zerocase`. If the exponent is equal to zero, we simply return the code of a function that, given any argument, returns 1: `box (abs (int 1))`.
  2. `ncase`. If the exponent is not zero, we first recursively construct the code for the exponentiation function for a smaller exponent (`reccall`). The result of this recursive call is a piece of code of type $\Box(Int \rightarrow Int)$. Then, this piece of code is un-boxed, and a new piece of code is constructed using the un-boxed value (`newbox`).

7.4 An Interpreter for $L_{\Box}$

We shall give the semantics of $L_{\Box}$ by providing an interpreter for the Omega encoding of the typing judgments of $L_{\Box}$.

The λ-fragment of $L_{\Box}$ is virtually identical to the interpreter for L_0 in Chapter 5. The important question is how to implement the modal fragment. In defining the meaning of $L_{\Box}$ programs, we are guided by the semantics of $\lambda^{\Box}$ described by Davies and Pfenning [31].

First, we must decide what meaning to give to expressions of type `Box a`. In a functional language (with recursion) the simplest meaning of boxed terms, as discussed by Davies and Pfenning, are suspended computations:

$$[\![\Box A]\!] = \mathbf{1} \to [\![A]\!]$$

Furthermore, such a semantics must respect the following identities [31, page 19]:

$$\begin{aligned} \text{box } e &= \lambda x : \mathbf{1}.\, e \\ \text{let box } u = e_1 \text{ in } e_2 &= (\lambda x : \mathbf{1} \to \tau.\, e_2[u := x\ ()])\ e_2 \end{aligned}$$

With these guidelines in mind, we can begin to devise an interpreter for $L_\Box$. The interpreter (Figure 7.6) takes a typing judgment of $L_\Box$ of type `(Exp d g t)`, a runtime modal environment, a runtime value environment, and returns a Haskell value of type `t`. As before, the runtime value environment is simply a value of type `g`. However, we have seen that for modal of type `t`, we use the type `() → t`, so the modal environment cannot simply be the nested tuple of type `d`. Rather, it is a closely related type `(ME d)`, defined below:

```
data ME~e = EMPTY
    | ∀e't. EXT (ME e') (() → t) where  e=(e',t)
      -- EXT      :: ME a -> (() -> b) -> ME (a,b)
```

Now, a runtime modal environment of type `ME ((),Int),Int)` can be created as follows:

```
me1 :: ((),Int),Int)
me1 = EMPTY 'EXT' (\ _ -> 1) 'EXT' (\ _ -> 2)
```

Finally, we are ready to give a type to the function `eval`:

```
eval :: Exp d g t → (ME d) → g → t
```

We concentrate on explaining the modal fragment (the bottom half of Figure 7.6):

1. Modal variables. The modal variable lookup is fairly standard. We consider the two relevant cases:

   (a) The **VarL** judgment provides us with an assumption that `gamma = (x,t)`. The runtime environment supplies another assumption, `gamma=(y,t2)`. These assumptions are combined to obtain an equality `t2=t`, which is induced by the type signature in relation to the result `(f ()::t2)`.

   (b) Similarly, the **ShiftL** case implements weakening. Again, assumptions introduced by pattern matching on the modal runtime environment are combined with the assumptions introduced by

```
eval                                :: Exp delta gamma t → ME delta → gamma → t
eval VarR e1 e2                     = snd e2
eval (ShiftR exp) e1 e2             = eval exp e1 (fst e2)
eval (Abs body) e1 e2               = (\v → (eval body) e1 (e2,v))
eval (App f x) e1 e2                = (eval f e1 e2) (eval x e1 e2)

eval VarL (EXT _ f) e2              = (f ())
eval (ShiftL e) (EXT env' _) e2     = eval e env' e2
eval (BoxExp body) e1 e2        = (Box (\ _ → (eval body e1 ())))
eval (UnBox expb body) e1 e2  =
    let ~(Box u) = eval expb e1 e2
    in eval body (ext e1 u) e2
```

Figure 7.6: The interpreter for $L_{\Box}$.

pattern matching over the $L_{\Box}$ judgment so that the weakened runtime modal environment can be passed as an argument to the recursive call of `eval`.

2. Box. First, we must decide how to represent *boxed* values. Here, we shall chose to define a data-type `Box a` as suspended computations over `a`. The `eval` function simply delays the evaluation of the body of the boxed expression and returns this computation wrapped up in a `Box`:

   ```
   data Box a = Box (() → a)

   eval (BoxExp body) e1 e2 = (Box (\ _ → (eval body e1 ())))
   ```

3. Unbox. The un-boxing is performed by first evaluating the expression to a `Box` value, binds the computation inside the `Box` in the 0-th position in the modal dynamic environment, and proceeds to evaluate the body of the `let box` expression.

   ```
   eval (UnBox expb body) e1 e2 =
       let (Box u) = eval expb e1 e2
       in eval body (ext e1 u) e2
   ```

It is worth reiterating the point made by Davies and Pfenning [31], that at first, there does not seem to be any difference between the meaning of the box modality, and simple call-by-name delay. While this is true, it is important to note that the modal type system of $L_{\Box}$ rejects certain programs that using delayed values (i.e., $() \to A$) would allow us to write. The type system accepts as correct only those programs that exhibit correct meta-programming properties (e.g., binding time separation [31]).

Chapter 8

Example: $\lambda^{\bigcirc}$

Davies and Pfenning define another version of the typed λ-calculus enriched with types based on temporal logic, called $\lambda^{\bigcirc}$. The logic on which the type system for $\lambda^{\bigcirc}$ is based is the discrete linear-time temporal logic.[1]

The motivation for devising this calculus seems to have been its ability to express, in a simple and natural way, binding-time analysis in partial evaluation [29]. The notion of "a particular time" in temporal logic correspond to computational stages (binding times) in partial evaluation.

b	$\in \mathbb{B} ::=$	$\mathsf{Int} \mid \cdots$	base types
τ	$\in \mathbb{T} ::=$	$\mathsf{b} \mid \tau \to \tau \mid \bigcirc\tau \mid \tau \times \tau$	types
Γ	$\in \mathbb{G} ::=$	$\langle\rangle \mid \Gamma, (\tau, n)$	type assignments
e	$\in \mathbb{E} ::=$	$c \mid 0 \mid e\uparrow \mid \lambda\tau.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e$	λ-fragment
	$\mid$	$\mathsf{next}\ e \mid \mathsf{prev}\ e$	temporal fragment

Figure 8.1: The syntax of the language $L_{\bigcirc}$.

8.1 Syntax of $L_{\bigcirc}$

The syntax of the language $L_{\bigcirc}$ is defined in Figure 8.1. The types of $L_{\bigcirc}$ are the types of the simply typed λ-calculus, enriched with $\bigcirc$-types. In logic, the formula $\bigcirc A$ indicates that A is valid at the next moment. Similarly, if we regard them as types of a programming language, we can see type $(\mathsf{Int} \to \bigcirc\mathsf{Bool})$ as a type of a function that takes an integer argument, and returns a boolean *at the next computational stage.* These computational stages are ordered with respect to evaluation, so that evaluation of all redices that occur at stage n happens before evaluation of the redices at the stage $n + 1$.

Type assignments are lists of types, where each type in a list is annotated with a natural number. This natural number represents the "time moment" (or stage) at which the free variable is bound.

[1] A "temporal logic is an extension to logic to include proofs that formulas are valid at particular times" [29].

$$\frac{}{\Gamma, (\tau, n) \vdash^n 0 : \tau}\text{Var} \quad \frac{\Gamma \vdash^n e : \tau}{\Gamma, (\tau', m) \vdash^n e \uparrow : \tau}\text{Shift}$$

$$\frac{\Gamma, (\tau_1, n) \vdash^n e : \tau_2}{\Gamma \vdash^n \lambda\tau_1.e : \tau_1 \to \tau_2}\text{Abs} \quad \frac{\Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash^n e_2 : \tau_1}{\Gamma \vdash^n e_1\ e_2 : \tau_2}\text{App}$$

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \mathsf{next}\, e : \bigcirc\tau}\text{Next} \quad \frac{\Gamma \vdash^n e : \bigcirc\tau}{\Gamma \vdash^{n+1} \mathsf{prev}\ e : \tau}\text{Prev}$$

Figure 8.2: Type System of $L_{\bigcirc}$.

The set of expressions consists of a completely standard λ-calculus fragment, and a temporal fragment consisting of two constructs:

1. next e. The *next* is an introduction construct for the circle types. Operationally, it delays the execution of the expression e until the next computational stage. In a way, it is analogous to the box expression of $L_{\Box}$, except that, as we will see, there is no requirement that e be closed.

2. prev e. The prev is en elimination construct for the circle types. While constructing a value at the next computational stage, the prev expression allows the control to pass back to the current stage, provided that its result is a next-stage value. This next stage value can then be plugged back into the next-stage context surrounding the prev.

8.2 Type System of $L_{\bigcirc}$

The typing judgment of $L_{\bigcirc}$ is defined in Figure 8.2. The typing relation $\Gamma \vdash^n e : \tau \subseteq \mathbb{G} \times \mathbb{N} \times \mathbb{E} \times \mathbb{T}$ is indexed by a natural number n, which represents a particular time at which an expression e has type τ. The typing rules for next and prev constructs manipulate this time index:

1. At some time index n, a value of type $\bigcirc\tau$ represents a value at the next moment. Thus, to show that next e has type $\bigcirc\tau$ at the moment n, we must prove that e has type τ at the time index $n + 1$.

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \mathsf{next}\, e : \bigcirc\tau}\text{Next}$$

2. An expression can be "escaped" by using prev only in the context of type-checking an expression at a later (non-0) point in time, and only if the escaped expression is a circle type (i.e., it already represents a computation at a later point in time). One should note that this formulation of the rule prevents typing of prev when the time index n is equal to zero, since there can be no earlier point in time.

$$\frac{\Gamma \vdash^{n} e : \bigcirc\tau}{\Gamma \vdash^{n+1} \mathsf{prev}\ e : \tau}\text{Prev}$$

The treatment of variables in the type system is also somewhat different from the simply typed λ-calculus. When a variable is bound by a λ-expression, the time index n at which it is bound is recorded in the type assignment together with the type of the variable. The variable rule is written in a way that ensures that only variables bound at time index n can be used at the same time index.

8.3 Encoding $L_{\bigcirc}$ in Omega

Recall that the typing judgments of $L_{\bigcirc}$ are indexed by a natural number that represents the time index at which the judgment is valid. Encoding this judgment as an Omega type constructor requires us to have a representation of natural numbers *at the level of types* in order to represent time indexes. Thus, we first define natural numbers at the type level, along the lines described in Chapter 4:

```
kind Nat                = Z
                        | S Nat
data IsNat (n :: Nat) = IsZero (Equal n Z)
                        | ∀m. IsSucc (Nat m) where n = (S m)
```

Natural numbers at the level of types are represented by the type constructors `Z` and `S` of kind `Nat`. The type constructor `IsNat n` is a runtime representation of the natural number `n`. The type signatures of the constructors are as follows:

```
IsZero :: Nat Z
IsSucc :: Nat n → Nat (S n)
```

The encoding of the typing judgment of $L_{\bigcirc}$ in Omega is shown in Figure 8.3. The type constructor `Exp` has three arguments:

1. The first argument, `n`, is the time index.
2. The second argument, `e`, is the type assignment. It is encoded as a nested tuple in the following mapping:

$$\begin{aligned} \mathsf{tr} &:: \mathbb{G} \rightarrow \mathsf{types} \\ \mathsf{tr}\ \langle\rangle &= () \\ \mathsf{tr}\ \Gamma, \tau^{n} &= (\mathsf{tr}\Gamma, (\tau, n)) \end{aligned}$$

```
data Exp (n::Nat) e t = ∀ e'.       Var     where e=(e', (t,n))
                      | ∀ e' t' m.  Shift   (Exp n e' t)   where e=(e',(t',m))
                      | ∀ t1 t2.    Abs     (Exp n (e,(t1,n)) t2)   where t=(t1 → t2)
                      | ∀ t1.       App     (Exp n e (t1 → t))   (Exp n e t1)
                      |             Const   t   String
                      |             Fix     (Exp n (e,(t,n)) t)
                      | ∀ t' m.     Next    (Exp m e t')   where t=(Circle n t'), m = (S n)
                      | ∀ m.        Prev    (Exp m e (Circle m t))   where n=(S m)
```

Figure 8.3: Typing judgment of $L_{\bigcirc}$ in Omega.

3. Finally, there is the representation of types. Base and arrow types of $L_{\bigcirc}$ are represented by their corresponding Omegatypes. The circle types are represented by the type constructor `Circle`, which we shall discuss in more detail later.

We examine the encoding of $L_{\bigcirc}$ judgments as the data-type `Exp` in more detail:

1. *The λ-calculus fragment.* The λ-calculus is fairly standard, except for the treatment of variables. First, in a λ-abstraction, a variable is bound at the same time index as the overall judgment. In the variable case, the time-index annotation in the type assignment is required to match the time-index of the overall expression.

2. *Next.* The 'next' construct is defined as follows. The argument to the constructor `Next` is an `Exp` of type `t'`, at the some time-index `m`. The equality constraint forces the type of the overall judgment, `t`, to be equal to `Circle n t'`. Finally, there is the additional equality constraint that `m` equals to `(S n)`. This forces the sub-expression argument to `Next` to be an expression at a higher time index.

3. *Prev.* The constructor `Prev` takes one argument: a sub-judgment of type `(Exp m e (Circle m t))`. There is also an equality proof that forces the overall judgment's time index `n` to be equal to the successor of `m`.

   It is worth noting how this prevents `Prev` expressions at time index zero. If we wanted to have an expression `Prev e` have the type `Exp Z e t` we would induce an equality proof obligation to show that `Z` equals `S m`, for some `m`. In Omega this would result in a type error.

The types of the constructors for the $L_{\bigcirc}$ judgments are listed in Figure 8.4. Let us look at a couple of simple examples of $L_{\bigcirc}$ judgments.

e1 :: Exp (S n) e (t → t)

```
Var   :: Exp n (e,(t,n)) t
Shift :: Exp n e t → Exp n (e,(t2,m)) t
Abs   :: Exp n (e,(t1,n)) t2 → Exp n e (t1 → t2)
App   :: Exp n e (t1 → t2) → Exp n e t1 → Exp n e t2
Next  :: Exp (S n) e t → Exp n e (Circle n t)
Prev  :: Exp n e (Circle n t) → Exp (S n) e t
```

Figure 8.4: Type signatures for constructors of $L_{\bigcirc}$ judgments.

```
e1 = Prev (next (abs var))
e2 :: Exp n e (Circle n t → Circle n t)
e2 = Abs (Next (Prev var))
e3 :: Exp n e (Circle n (t1 → t2) → Circle n t1 → Circle n t2)
e3 = Abs (Next ((Prev (Shift Var)) `App` (Prev Var)))
```

1. The judgment `e1` is "escaped," using `Prev` at the top level, so the Omega type checker infers `(S n)` as its time index.

2. The judgment `e2` is an identity function that takes an argument of type $\bigcirc\tau$ and immediately splices it, using `Prev` into a `Next`-stage code. The `Next` and `Prev` cancel each other out, leaving an identity function of type $\bigcirc\tau \rightarrow \bigcirc\tau$.

3. The judgment `e3` is slightly more complicated. It takes two arguments, a function $\bigcirc(\tau_1 \rightarrow \tau_2)$ and a delayed value of type $\bigcirc\tau_1$, and produces a delayed result of type $\bigcirc\tau_2$.

8.4 An Interpreter for $L_{\bigcirc}$

In defining an interpreter for $L_{\bigcirc}$ we are guided by the big-step semantics for a small temporal functional language defined by Davies and Pfenning [29]. They define the semantics of this language as a family of functions, indexed by a natural number representing the time index, which maps expressions to values (written:) $e \overset{n}{\hookrightarrow} v$.

The interpreter we define here is based on the same idea, although it has a more denotational style. The following observations can be taken as general guidelines in defining the interpreter.

Time-indexed evaluation

The work that the interpreter performs can be divided into three distinct modalities, based on the time index.

1. At the time index 0. The time index 0 represents expressions that are to be evaluated *now*. This means that the λ-calculus fragment must be interpreted at the time index 0. For example, at time index 0, should map the expression $(\lambda.\mathsf{Var})0$ to the integer value 0, and so on.

2. At the time index 1. The time index 1 represents expressions that are to be evaluated at the next stage (i.e., the next moment in time). In particular, this means that the real work (e.g., reducing β-redices) of the λ-calculus fragment is not to be performed at time index 1. However, escaping expressions of the form ($\mathsf{prev}\ e$) can occur inside time index 1 expressions. In this case, the expression e must be evaluated *at time index 0*, produce a time-index-1 value that is to be spliced in place of $\mathsf{prev}\ e$.

   This is illustrated in Davies and Pfenning's big-step semantics by the following rule:

$$\frac{e \overset{0}{\hookrightarrow} \mathsf{next}\ v}{\mathsf{prev}\ e \overset{1}{\hookrightarrow} v}\text{eval1}$$

3. At the time index $n > 1$. At the time index greater than 1, there is no real work. The interpreter must merely traverse, and rebuild, the original term, making sure to increment its time index when evaluating under `next`, and to decrement its time index when evaluating under `prev`.

Values

The interpreter for $L_{\bigcirc}$ is a written as a family of functions indexed by a natural number presenting the time index. It must well-typed expressions (judgments) of $L_{\bigcirc}$ into *values*. At the time index 0, the values for the λ-calculus fragment seem quite straight-forward: an expression of type `Int` $\rightarrow$ `Int` can simply be mapped into an `Int` $\rightarrow$ `Int` function. However, when considering the modal fragment, the notion of values gets a little more complicated.

First, at the time index 0, we have a type of values `Circle n t` that represent the delayed (modal) values of type `t` at time index `n`. Second, Davies and Pfenning introduce a notion of a set of values, at some index n, that is a subset of the set of expressions in a particular normal form. The idea is that the set of values at time index $(n + 1)$ is isomorphic to the set of expressions at time index 0.

```
data Val n e t =
          ValConst t
| ∀ m e'. VarV where e = (e',(t,n)), n=S m
| ∀ m p t2 e'.    ShiftV (Val (S m) e' t)
         where e = (e',(t2,p)), n=(S m)
| ∀ m t1 t2.  AbsV (Val (S m) (e,(t1,(S m))) t2)
              where t=t1->t2, n = S m
| ∀ m t1.AppV (Val n e (t1->t)) (Val n e t1) where n = S m
```

```
| ∀ t'.  NextV (Val (S n) e t') where t = (Circle n  t')
| ∀ m t1.PrevV  (Val (S m) e (Circle (S m) t)) where n =  (S (S m))
| ∀ env. Closed env (Val n env t)

data Circle n t = ∀ env. Circle env (Val (S n) env t)
```

8.4.1 The Interpreter

With this in mind, we can tentatively assign a type to the interpreter. In order to be able to tackle the three distinct interpreter modes in separate steps, we shall divide the interpreter into three functions:

```
eval0 :: Exp Z e t→e→t
eval1 :: Exp (S Z) e t→e→Val (S Z) e t
evalN :: IsNat n→Exp (S n)→e→Val (S n) e t
```

First, we present the interpreter at time index 0. The λ-calculus fragment is fairly standard (see Chapter 5).

```
1 eval0 :: Exp Z e t→e→t
2 eval0 (Const c t) env = c
3 eval0 Var (env',(v,Z)) =  v
4 eval0 (Shift e) (env',_) =  eval0 e env'
5 eval0 (Abs e) env = \x→eval0 e (env,(x,Z))
6 eval0 (App e1 e2) env = (eval0 e1 env) (eval0 e2 env)
7 eval0 (Next e) env = Circle env (eval1 e env)
```

The only exception is in the treatment of variables.The values in the runtime environment carry their time indexes. These time indexes are ignored when extracting values from the environment (lines 3 and 4). The λ-abstraction case must bind a new variable in the runtime environment (line 5). In addition to the actual value, its time index (`Z`) is also bound.

Let us consider the modal fragment. The first thing to note is that the function `eval0` is not defined for the case when the $L_{\bigcirc}$ judgment is of the form `Prev e`. This is because, by definition of `Exp`, judgments of the form `Prev e` cannot have the type `Exp Z e t`. Finally, on line 7, we show the definition of `eval0` for the judgment of the form `Next e`. First, the sub-expression `e` of type `Exp (S Z) e t` is evaluated by `eval1`, to obtain the result of type `Val (S Z) e t`. Such a value, together with the current environment `env` can be wrapped inside a `Circle` value to obtain the result of type `Circle (S Z) t`.

Now, let us consider the definition of `eval1`, the interpreter at the time index 1:

```
8  eval1 :: Exp (S Z) e t→e→Val (S Z) e t
9  eval1  Var env =  VarV
10 eval1 (Shift e) env = ShiftV (eval1 e (fst env))
11 eval1 (Abs e) env = AbsV (eval1 e (env,(undefined,undefined)))
12 eval1 (App e1 e2) env = AppV (eval1 e1 env) (eval1 e2 env)
13 eval1 (Prev e) env = case (eval0 e env) of
14                       Circle e val→Closed e val
15 eval1 (Next e) env =  NextV (evalN two e env)
16   where two = IsSucc (IsSucc IsZero)
```

For the λ-calculus fragment of $L_{\bigcirc}$, the function `eval1` performs *rebuilding*. The simplest example of this is on line 9: starting with the variable expression `Var :: Exp (S Z) (e,(t,S Z))`, it constructs a value VarV :: Val (S Z) (e,(t,S Z)). For other λ-fragment expressions (lines 10-12) such rebuilding is performed recursively on the structure of the term.

The most interesting part of `eval1` is the case for `Prev` judgments (line 13-14). First, the sub-judgment `e` is evaluated by `eval1` to obtain a circle (delayed) value. This value is de-constructed, its `Val` judgment extracted. The actual splicing of this code is performed by the constructor **`Closed`**, which allows us to form a *closure* out of any value, by remembering the environment in which it is defined.

The case of `Next e` (line 15) proceeds by evaluating the judgment `e` at a higher time index to obtain a value of type `Val (S (S Z)) e t`, and then wrapping the result with `NextV` to obtain a value of type `Val (S Z) e (Circle (S Z) t)`.

Finally, we consider the function `evalN` which implements the interpreter at a time index greater than 1:

```
17 evalN :: IsNat (S (S n))→Exp (S (S n)) e t→e→Val (S (S n)) e t
18 evalN (IsSucc (IsSucc n)) Var env = VarV
19 evalN (IsSucc (IsSucc n)) (Shift e) env =
20       ShiftV (evalN (IsSucc (IsSucc n)) e (fst env))
21 evalN (IsSucc (IsSucc n)) (Abs e) env =
22       AbsV (evalN (IsSucc (IsSucc n)) e (env, (undefined,undefined)))
23 evalN (IsSucc (IsSucc n)) (App e1 e2) env =
24       AppV (evalN (IsSucc (IsSucc n)) e1 env)
25            (evalN (IsSucc (IsSucc n)) e2 env)
26 evalN (IsSucc (IsSucc (IsSucc n))) (Prev e) env =
27       PrevV (evalN (IsSucc (IsSucc n)) e env)
28 evalN  n (Next e) env = NextV (evalN (IsSucc n) e env)
```

The function `evalN` takes as its first argument a natural number representation of the current time index. To ensure that this time index is at least 2, the argument's type is specified as `IsNat (S (S n))`. In both the λ-calculus and the temporal fragment the function `evalN` behaves the same: the judgments are recursively rebuilt (transformed into values), while the time indexes increment and decrement whenever `Next` or `Prev` is encountered.

8.4.2 Power Function

The first example we present is that of the `power` function, analogous to the one shown in Figure 7.5 for $L_{\Box}$.

```
1  power :: Exp Z env (Int→(Circle Z Int)→(Circle Z Int))
2  power = Fix (Abs (Abs (Abs body)))
3    where body  = myif ‘App‘ v1 ‘App‘ v0 ‘App‘ body2
4          body2 = Next (times ‘App‘ (Prev v0) ‘App‘
5                           (Prev
6                            (v2 ‘App‘ (minus ‘App‘ v1 ‘App‘ one) ‘App‘ v0)))
7          myif  = Const (\c t e→if c then t else e) "if"
8          one   = Const 1 "one"
9          times = Const (+) "+"
10         minus = Const (-) "-"
11
12 example = Next (Abs (Prev (power2 ‘App‘ (Const 2 "2") ‘App‘ (Next Var))))
13 -- Next (\x→(Prev (power 2 (Next x))))
14 result  = eval0 example ()
```

The function `power` takes two arguments. The first, the exponent is an integer value. The second, the base, is a delayed integer value (of type `Circle Z Int`), and produces as a result a delayed integer value (of type `Circle (S Z) Int`). The function power can be specialized (line 12) to exponent two to obtain a delayed function value of type `Circle Z (Int→Int)`. Evaluating `example` (line 14) yields the following result (slightly cleaned-up and pretty-printed):

```
result =
(Circle
   (AbsV
      (AppV
         (AppV
            (ValConst <fn> "times")
            (AppV (AppV (ValConst <fn> "times") (ValConst 1 "1"))
                  VarV))
```

```
VarV))) : Circle Z (Int -> Int)
```

Part IV

Conclusion

Chapter 9

Related Work

We shall organize our survey of related work by dividing it into 2 broad topics:

1. *Meta-programming.* (Section 9.1)
   (a) *Homogeneous meta-programming.* (Section 9.1.1)
   (b) *Heterogeneous meta-programming.* (Section 9.1.2)
2. *Dependent types, type theory and meta-programming.* (Section 9.2)
   (a) *General background.* (Section 9.2.1)
   (b) *Meta-programming with dependent types.* (Section 9.2.2)
   (c) *Simulating dependent types in Haskell.* (Section 9.3)

9.1 Meta-Programming

Here we provide a very general overview of the work most directly relevant to this dissertation. We begin with some background remarks on meta-programming, noting that a more detailed historical and taxonomic survey of programming languages that support meta-programming has been written by Sheard [118].

The notion of treating programs as data was first explicitly developed by the LISP community. In this context, the notion of *quasi-quotation* [126, 8] was developed as a way of making the interface to the data representing the object program "as much like the object-language concrete syntax as possible." [118] A historical discussion, tracing quasi-quotation from the original ideas of Quine, to their impact on MetaML is given by Taha [128]. The idea of the need for a meta-language (that can be used as a common medium for defining and comparing families of (object) languages) can be traced to Landin [69]. Similarly, Böhm proposed using the λ-calculus-based language CuCh as a meta-language for formal language description [12].

Nielson and Nielson [90, 93, 92] define programming languages and calculi that clearly distinguish meta-level from object-level programs as a part of the language. This work can be seen as motivated by a search for a formal way to study the semantics of compilation. They recognized that compilation can be seen a meta-program with two phases: a static phase, where the compiler constructs a residual output program from some input program, and a dynamic phase where the residual program itself is executed. Thus, they design a functional language with two levels corresponding to the two phases of compilation [94]: all language constructs come in two flavors, minimally distinguished by the syntax. We note also that the two levels are essentially the same language, i.e., that the meta-programming described is homogeneous. Nielson and Nielson study the denotational semantics of such two-level languages, [91] as well as their applications to abstract interpretation [92]. They also generalize their work to multi-stage languages [95].

An important impetus to the study of meta-programming languages came from the partial evaluation community.

Partial evaluation researchers approached the problem from a more syntactic point of view, not really considering the staging constructs as first-class (semantically motivated) parts of the language. With the benefit of hindsight, however, this perhaps explains why they did not develop type systems that would statically guarantee type correctness of both the static and dynamic stages in two-level languages.

Gomard and Jones [49] present a two-level λ-calculus as a part of their development of a partial evaluator for the λ-calculus and the study of binding time analysis for such an evaluator. In this scheme, a binding time analyzer takes a (single-level) λ-expression, and produces a two-level λ-expression. Then, the semantics of the two-level calculus can be used to reduce 2-level expressions produced by the BTA, yielding a residual program that consists entirely of the level-2 parts of the 2-level expression. They also develop a type system for the 2-level calculus in order to be able to judge the correctness of the annotations produced by the BTA. However, only level-1 terms are typed; the residual programs constructed using the dynamic part of the 2 level calculus are dynamically typed.

Glück and Jørgensen [46, 47] studied binding time analysis and partial evaluation with more than two stages. Their generalization of binding time analysis to multiple stages is acknowledged [128] as being a major source of inspiration for the MetaML family of multi-stage languages.

Two important meta-programming systems emerged from the study of constructive modal logic by Davies and Pfenning [30, 29] (See Section 9.1.1).

MetaML [137] (See Section 9.1.1 for a detailed discussion) is an important synthesis of many previous *generative meta-programming languages*. It extends the work on modal calculi of Davies and Pfenning, introducing new concepts such as cross-stage persistence, and type-safe combination of reflection (the *run* construct) with open code.

9.1.1 Homogeneous Meta-Programming

The division into homogeneous and heterogeneous meta-programming languages has was introduced by Taha [128] and Sheard [118, for an excellent survey]. In this section, we shall trace the context of homogeneous meta-programming, starting with modal-logic based λ-calculi, and proceeding to MetaML.

Modal Logic: $\lambda^{\Box}$ and $\lambda^{\bigcirc}$

Many homogeneous meta-programming systems are motivated by the study of modal logic. In particular, we shall examine two related logical systems (and their associated versions of the λ-calculus): the first, $\lambda^{\Box}$, corresponds to the modal logic $S4$; the second, $\lambda^{\bigcirc}$, corresponds to linear-time temporal logic. Both of these systems have found applications in the study of meta-programming. Each of them captures an important intuition about program generators. One of them, $\lambda^{\Box}$ captures the notion of *closed code*, which can be executed from within the meta-program. The other, $\lambda^{\bigcirc}$, allows manipulation of open code fragments that can be easily combined by "splicing." Combining the two modalities results in a meta-programming language that captures very precisely the notion of homogeneous program generators. However, such a combination is not straightforward, since the splicing (escaping) of $\lambda^{\bigcirc}$ and code execution (run) of $\lambda^{\Box}$ interact and interfere with each other. Sheard and Thiemann provide a good discussion of the issue and a survey of related work that addresses it [125].

The calculi $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ both use modal operators, $\Box$ (necessity) in $\lambda^{\Box}$ and $\bigcirc$ (next) in $\lambda^{\bigcirc}$, to classify terms that produce object-language programs. For example, the type $(A \rightarrow \Box B)$ is seen as a type of a program generator that takes an argument of type A and produces an object language program of type B.

The calculus $\lambda^{\Box}$ can be seen as the language of proof terms for the propositional modal logic $S4$ [108]. On the logical side, box ($\Box$) is the necessity operator. The necessity operator corresponds to a type of *code*, i.e., values that represent object language expressions. In particular, it classifies, a type of *closed code*, i.e., a type of object programs that do not contain free variables. The type system of $\lambda^{\Box}$ ensures that no free variables escape from the **box** construct by keeping two type assignments, Δ, and Γ, (see Figure 9.1). When type-checking the expression **box** e, the expression e must be type-checked, in the empty the type assignment Γ (this is the type assignment that is augmented when type-checking λ-abstractions), indicating that there are no free variables in e. However, the box elimination construct binds its variable in the *other* environment(Δ) thus allowing manipulation of unboxed values when building inside other boxed expressions.

Let us consider a standard example, the `power` function which, given two integers n and x, computes x^n. Rather than providing a function of type $\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$, we shall define a function of the related type $\texttt{Int} \rightarrow \Box(\texttt{Int} \rightarrow \texttt{Int})$, i.e., given the argument n, it produces a *program* that when given the argument

λ-fragment

$$\frac{}{\Delta;\Gamma,x:\tau\vdash x:\tau}\text{Var1}\quad\frac{}{\Delta,x:\tau;\Gamma\vdash x:\tau}\text{Var2}\quad\frac{\Delta;\Gamma\vdash x:\tau}{\Delta,y:\tau';\Gamma\vdash x:\tau}\text{Weak1}\quad\frac{\Delta;\Gamma\vdash x:\tau}{\Delta;\Gamma,y:\tau'\vdash x:\tau}\text{Weak2}$$

$$\frac{\Delta;\Gamma,x:\tau_1\vdash e:\tau_2}{\Delta;\Gamma\vdash\lambda x:\tau_1.e:\tau_1\to\tau_2}\text{Abs}\quad\frac{\Delta;\Gamma\vdash e_1:\tau_1\to\tau_2\ \Delta;\Gamma\vdash e_2:\tau_1}{\Delta;\Gamma\vdash e_1\ e_2:\tau_2}\text{App}$$

$\Box$-fragment

$$\frac{\Delta;\langle\rangle\vdash e:\tau}{\Delta;\Gamma\vdash\mathsf{box}\ e:\Box\tau}\text{Box}\quad\frac{\Delta;\Gamma\vdash e_1:\Box\tau_1\quad\Delta,x:\tau_1;\Gamma\vdash e_2:\tau}{\Delta;\Gamma\vdash\mathsf{let\ box}\ x=e_1\ \mathsf{in}\ e_2:\tau}\text{Unbox}$$

Figure 9.1: The type system of $\lambda^{\Box}$.

x computes x^n.

```
power      :: Int → □(Int → Int)
power 0     = box (\x → 1)
power (n+1) = let box f = power n in box (\x → x * (f x))
```

Applying the power function to the argument 2 yields the program `pow2`. Note that `pow2` contains a number of "administrative redices." The generated program can be run by using **let box** construct to obtain the value of 3^2, shown below.

```
pow2 :: □(Int → Int)
pow2 = power 2
--  pow2 = box(\x1 → x1 * ((\x2 → x2 * ((\x3 → 1) x2)) x1))

result = let box f = pow2 in f 3
--  result = 9
```

Davies and Pfenning [29] also studied type systems extending the Curry-Howard isomorphism from simple propositional logics to the constructive (linear-time) temporal logic. Such a system is shown to accurately describe binding time analysis in (multi-stage) partial evaluation. More notably, they state and prove the property of *time-ordered normalization*. This property means that reductions preserve the binding time of redices in $\lambda^{\bigcirc}$ terms: all terms typed at an earlier "time", say $\bigcirc A$ are evaluated before terms typed at a later time, say, $\bigcirc\bigcirc A$. This property also means that $\lambda^{\bigcirc}$ realistically describes partial evaluation, or, more generally, generative multi-staged meta-programming of a certain kind.

λ-fragment

$$\frac{}{\Gamma, x^n : \tau \vdash^n x : \tau}\text{Var} \quad \frac{\Gamma \vdash^n x : \tau}{\Gamma, y : \tau'^m \vdash x : \tau}\text{Weak}$$

$$\frac{\Gamma, x : \tau_1^n \vdash^n e : \tau_2}{\Gamma \vdash^n \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\text{Abs} \quad \frac{\Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \; \Gamma \vdash^n e_2 : \tau_1}{\Gamma \vdash^n e_1 \; e_2 : \tau_2}\text{App}$$

$\bigcirc$-fragment

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \mathsf{next}\ e : \bigcirc\tau}\text{Next} \quad \frac{\Gamma \vdash^n e_1 : \bigcirc\tau}{\Gamma \vdash^{n+1} \mathsf{prev}\ e_1 : \tau}\text{Prev}$$

Figure 9.2: The type system of $\lambda^{\bigcirc}$.

They prove that their calculus is equivalent with the system of Gomard and Jones [49] by providing translations between them.

The main technical trick in the type system (Figure 9.2) is

1. To annotate the typing judgment with a natural number *level* index. This index is augmented when type-checking inside the next construct, which delays evaluation. Similarly, the index is decremented when type-checking the prev construct, which escapes back to the previous level to compute a program fragment that is to be spliced into a larger object-program context.

2. To annotate the variable bindings in the type assignment Γ with the level at which those variables are bound. This assures that no variable in the program can be used "at the wrong time," thus preventing *phase errors*, situations in which a variable is used before it is defined.

The `power` function example can be replicated in $\lambda^{\bigcirc}$ as well.

```
power      :: Int → ◯Int → ◯Int
power 0     x = next 1
power (n+1) x = next (( prev x) * ( prev (power n ( next x))))

result = power 2 (next 3)
 -- result = next (3*3*1)
```

Note that the residual code produced by the $\lambda^{\bigcirc}$ version of the `power` function does not contain the extraneous β-redices present in the residual code generated by the $\lambda^{\Box}$ implementation.

Type systems of $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ are interesting examples of non-standard type systems. In Chapters 7 and 8 we describe encodings, using our Haskell techniques, of well-typed terms in $\lambda^{\Box}$ and $\lambda^{\bigcirc}$, as well as well-typed interpreters for a version of both languages. It is interesting to note that in our $\lambda^{\bigcirc}$ something very much like the time-ordered normalization (see above) becomes a statically enforceable invariant encoded in the type of an interpreter for the encoding of well-typed $\lambda^{\bigcirc}$ terms.

MetaML

The calculi based on $\Box$ and $\bigcirc$ modalities have both comparative advantages and disadvantages when used in meta-programming. The program generators written in $\lambda^{\bigcirc}$ tend to be easier to write and generate more efficient residual programs. Program generators written with $\lambda^{\Box}$ tend to leave a large number of "administrative redices" in the residual programs [29]; some of these administrative redices can be eliminated in $\lambda^{\bigcirc}$.

The disadvantage of $\lambda^{\bigcirc}$ is that the generated residual code (whose types are classified by the $\bigcirc$ type constructor) cannot be programmatically executed in the type system of $\lambda^{\bigcirc}$ (there is no $\lambda^{\bigcirc}$ analogue to unbox).[1]

The considerable body of research on MetaML [135, 130, 82, 15, 129, 134] is an attempt to combine the ease of programming of the $\bigcirc$ modality with the ability to run generated programs of the $\Box$ modality, all in a strongly typed setting. The general approach can be outlined as follows:

1. MetaML uses a slightly modified version of the $\bigcirc$ modality. In MetaML, it is a type constructor, called "code", and written as a bracket around a type: $\langle A \rangle$. The constructs prev and next are replaced by bracketed code templates ($\langle e \rangle$) and escapes ($\tilde{e}$).

2. MetaML introduces a *run* construct in the language which takes an expression of type $\langle A \rangle$ and produces an expression of type A. However, this is unsound in general, since $\langle A \rangle$ might contain free variables whose value bindings may not be known at the time of running the piece of code. A number of type-systems have been devised to deal with this problem:

   (a) Before a piece of code can be run, the type system must prove that it is closed [135]. This is done by making sure that it is typable in an empty typing context. While this approach is safe, there are situations in which it rejects programs that are perfectly safe. It also prevents λ abstractions over certain terms that contain *run*.

[1] In practice this means that execution of residual programs generated is performed by some extra-linguistic (ad hoc) means – e.g., a top-level way of executing programs.

(b) Counting the number of escapes, brackets and run that surround a term can also used to prevent run from going wrong. [130] This is a very syntactic method of ensuring the safety of run. Similarly, such a type system can be seen as being to restrictive, disallowing abstractions over *run*. Nevertheless, this type system can be quite useful in practice. The programming language implementation of MetaML [121] uses a type system based on this idea, extended to a conservative extension of Standard ML of New Jersey [79].

(c) A further development of MetaML type system, called AIM [82], solved this problem by introducing an additional modality, essentially "box," as a refinement of the MetaML type of code. Then, only boxed code terms ($[\langle A \rangle]$) can be executed using *run*. Unlike the previous approaches, this type system allows for abstractions over *run*. However, the formalism of AIM makes the meta-programming with explicit staging a lot more verbose and, at times, somewhat awkward [134, for discussion].

(d) Recently, Taha and Nielsen presented a type system for MetaML using *environment classifiers* [134]. The environment classifiers are a formal way explicitly naming environments in which free meta-variables appearing in open code are defined. The advantage of this type system is that is allows the safe kinds of open code to be executed. This approach harmonizes the tension between the approaches (2b) and (2c): "while the first approach allows us to run open code, but not abstract *run*, the second allows us to do the latter but not the former. [The type system with environment classifiers] provides both features." [134, page 2].

(e) Finally, addressing the same difficulties of safely combining *run* and *escape* as (2d), Sheard and Thiemann [125] design another type system for MetaML. This type system is based on constraint solving and subtyping. The advantage of this system seems to be that it does not require the programmer to supply any annotations beyond the usual MetaML ones, and that it seems amenable to type inference. Unlike Taha and Nielsen's type system [134], it does not require explicit annotations on cross-stage persistent constants.

In addition to these features, MetaML supports *cross-stage persistence*, allowing later-stage programs to use values defined at an earlier stage. A good deal of work has also been done to support staging of imperative MetaML programs [15], as well as to provide "industrial strength" implementations of MetaML [16, 121].

We digress, briefly, to consider how is MetaML, a homogeneous meta-language, relevant to the heterogeneous meta-programming framework we propose. In the examples presented throughout this dissertation, we have concentrated on heterogeneous programming scenarios of a particular kind. In this kind of heterogeneous meta-programming, we use a homogeneous fragment of the meta-language as an efficient "back

end" for implementation. The scenario can be described as follows:

1. We start with an encoding of the syntax (and type system) of some object language L_1.
2. We manipulate the object-programs by deconstructing their representations and (e.g., in the case of interpreters) map them to some domain of values, also encoded as a part of the meta-language.

However, we can use staging in step 2, obtaining not a value in the meta-language, but a residual program in the meta-language that computes that value. The usual arguments for obtaining more efficient programs through staging still apply in this situation. This way, we have a heterogeneous system which translates programs in the object language L_1 into programs (represented in MetaML-style using the code type) in the meta-language. By so combining heterogeneous and homogeneous meta-programming we can reap the benefits from both worlds: we can (a) safely manipulate object programs in many different object languages, while (b) writing highly efficient interpreters for such object languages by using MetaML notion of staging to remove the interpretive and tagging overhead inherent in writing interpreters for such object languages.

9.1.2 Heterogeneous Meta-Programming

Now, we shall briefly trace the genealogy of the main ideas presented in this dissertation. A couple of additional topics is worth mentioning in connection to heterogeneous meta-programming – intentional analysis, and the pragmatics of the interface to object-language syntax – and we shall review them.

A Historical Overview

Initial motivation for our study of heterogeneous meta-programming came from the work on implementing of domain specific languages in a safe, efficient and disciplined way by staging interpreters in MetaML [120]. The basis of this approach is to define an object language as a data-type in MetaML [121], write a definitional interpreter for it, and then stage this interpreter to obtain an efficient residual program from which the interpretive overhead has been removed.

The first problem, however, was that with algebraic data-types in MetaML there was no way of ensuring that only well-typed programs of the object language are interpreted. If the object language is strongly typed, developing a way of statically ensuring well-typedness of the object language encoding would provide an additional sense of safety (and reliability) of staged interpreters by guaranteeing that no type errors would be generated by the residual program. Furthermore, we saw that encoding and using type information about the object language would allow us to generate *tagless staged interpreters* that are more efficient because no tagging overhead is introduced by the implementation [102].

As a first step we developed a prototype meta-language with staging and dependent types [102], and implemented our first tagless staged interpreters. This language was initially modeled on Coq [139] and similar type theory-based languages. However, it was soon recast into a FLINT-style [116] framework that allowed us to write staged interpreters in a meta-language with effects such as general recursion and partiality without having to compromise either type safety of the meta-language, or the expressive power to represent well-typed object-language terms.

Such work provided a proof of concept and a simple prototype implementation, but we were concerned with a couple of pragmatic issues. First, implementing and, more importantly, maintaining a large new programming language with a complicated type system and staging did not seam feasible at the time. Second, existing programming languages with dependent types [2] did not seem to attract a large user base among functional programmers.

At the same time, we became aware of Baars and Swierstra's [4] paper that used equality types to represent types at runtime as a way of integrating dynamic typing in Haskell. Their work, in turn, has roots in Weirich's paper presenting an encoding of equality between types in Haskell [143]. Also relevant is McBride's work on simulating dependent types in Haskell [75].

We adapted these techniques to represent typing judgments of λ-calculus terms in Haskell. To do this we needed to use only very standard extensions to Haskell98, available most Haskell implementations, such as higher-rank polymorphism and existential types. To experiment with staging, we assumed that Haskell can be extended, conservatively, with staging constructs [2]. In this programming language environment, we were able to define the same tagless staged interpreters, and apply our technique to a larger set of heterogeneous meta-programming examples.

Intentional Analysis

Here, we take *intentional analysis* to refer to the ability of a meta-program to analyze the (syntactic) representation of object-programs. In the context of homogeneous meta-programming systems such as MetaML, intentional analysis is problematic from the semantic point of view. Indeed, MetaML (and related systems) are known as *generative meta-programming languages* since the programmer is only allowed to generate programs, not to rewrite or examine them.

MetaML and Intentional Analysis. MetaML enjoys interesting (non-trivial) equational properties. The $\alpha-$,$\beta-$, $\eta-$, and bracket-cancellation reductions are sound with respect to the operational semantics of MetaML [129]. This allows the MetaML implementations to perform a number of optimizations on their

[2] An implementation of such a language was produced by Sheard [119, for Tim Sheard's prototype implementation].

representation of code without changing the meaning of programs. For example, certain trivial β-redices are removed, rewriting simplifications based on monadic laws are performed on code containing Haskell-style `do` expressions, and `let` expressions are hoisted to avoid excessive nesting.

All these optimizations yield a representation of code that is highly readable when printed. Automatic removal of certain administrative redices in code representation sometimes also yields more efficient object programs. However, the equational properties that justify these optimizations conflict with intentional analysis. Simply put, if the user can observe the difference between, say the pieces of code $\langle$`(fn x => x) 1`$\rangle$ and $\langle$`1`$\rangle$, then the equational properties are no longer sound and optimizations cannot be safely performed. A satisfactory formulation of intentional analysis that can be safely integrated into MetaML implementations has yet to be discovered.

In the techniques for heterogeneous meta-programming we propose, intentional analysis can be performed on source object languages encoded by the programmer (while still statically ensuring that such intentional analysis preserves typing properties of the object program). The part of the meta-language that deals with staging, however, allows only generative meta-programming. We conjecture that in practice this will prove to be a reasonable compromise: intentional analysis can be used to perform optimizing source-to-source transformations on the syntax of object-language programs, while staging is used to ultimately ensure efficiency of object-language implementations.

FreshML. Pitts and Gabbay [39, 41, 111] formulate an elegant theory for manipulating abstract syntax with the notion of variable binding. From the programming language point of view, this allows them to construct data-types representing sets of syntactic terms modulo α-conversion. Theoretical foundation of this work is Frænkel-Mostowsky set theory, which provides models for such sets of terms. Unlike previous approaches, such data-types admit a simple and elegant notion of structural induction, while still preserving α-equivalence.

Integrating the key ideas of FreshML and nominal logic into a meta-programming framework has already been proposed by Nanevski [86]. In our examples, we opt for de Bruijn style of representing syntax modulo α-renaming. The main reason is that it is not entirely clear how to express typeful abstract syntax in this framework, although we conjecture that Nanevski's scheme might very well be adapted to our encoding of typing judgments. The investigation of this question is left for future work.

Names and necessity. An interesting approach to meta-programming was proposed by Nanevski and Pfenning [86, 85]. It can be seen as a parallel effort to solve many of the same problems that MetaML was invented to address. Whereas MetaML starts with the $\bigcirc$ modality and finds various ways of relaxing it to allow for execution of open code, Nanevski's language, $\nu^{\Box}$ starts with the $\Box$ modality and relaxes

its restrictions on open code by using the ideas form Pitts and Gabbay's nominal logic to allow certain kinds of free variables. The main idea seems to be to combine $\lambda^{\Box}$ of Davies and Pfenning with nominal logic of Pitts and Gabbay [40]. This nominal-modal logic seems to provide a very rich system for meta-programming: the modal fragment allows for construction of programs, including the run operation, while the nominal fragment permits certain kinds of intentional analysis over constructed meta-programs. Thus, while residual programs generated in this framework tend to be very similar to ones generated by $\lambda^{\Box}$, intentional analysis can be used to make the residual programs considerably more efficient.

DALI. For sentimental reasons, we mention an attempt by the author to study a form of intentional analysis in the context of the λ-calculus [100]. The approach was inspired by a proposal by Dale Miller [78] for extending data-types in ML-like languages with a form of higher-order abstract syntax. Our approach was to introduce a kind of object-level bindings that can be deconstructed using a form of higher-order patterns in a λ-calculus. We studied the reduction semantics of such a calculus in an untyped setting and showed its coherence with a rather standard natural semantics. This approach has been superseded by the considerably more elegant theories of Pitts and Gabbay's nominal logic [40] and $\nu^{\Box}$ [86].

Typeful code representation. Hongwei Xi and Chiyan Chen [18] have presented a framework for meta-programming (though they seem mostly interested in studying homogeneous meta-programming) that is very similar to the approach described in Chapter 5. This work seems to have been carried out synchronously with our work, and we became aware of it relatively late in the course of our own investigations.

Xi and Chen represent object-language programs using constants whose types are essentially the same as the smart constructors for the type `Exp e t` in Chapter 5. Instead of Haskell, they use their own language with guarded recursive data-type constructors [146]. The use of guarded recursive data-types makes it unnecessary to resort to equality-proof based encodings that we use in our implementations, and thus results in code that is both easier to read and write. The main difference between their examples and ours is that we show how we can combine the use of staging with typeful syntactic representations to derive more efficient implementations.

Finally, they present embeddings of MetaML into this language by translation. This translation, however, seems to be a meta-theoretical operation which is not defined in the language itself. Rather, they seem to see their language as a general semantic meta-language. Other meta-programming languages like $\lambda^{\Box}$, $\lambda^{\bigcirc}$ and MetaML can be given semantics by translation into their language. It might be interesting to compare our implementations of $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ in Chapters 7 and 8 to their translations.

Pragmatics of Object-language Syntax

In the examples presented in this dissertation, we use algebraic data-types (albeit augmented with techniques that allow them to statically encode important program invariants) to represent the syntax of object-language programs. Parsing and pretty-printing interface to these data-types can be implemented. This is done *post hoc*, by writing functions that construct elements of these algebraic data-types from a purely syntactic representation (e.g., strings or untyped s-expressions). In Chapters 3 and 5, we have shown how to write such functions (called `typeCheck`).

In this dissertation, we chose not to concentrate further on this problem since a simple, though not very practical, solution for it does seem to exist. However, a heterogeneous meta-programming language would gain considerably in usability, if the programmer could be exposed to object-language programs through some kind of interface based on concrete syntax of the object-language. Concrete types, or *conctypes* [1] are a way of allowing the programmer to specify her algebraic data-types in a BNF-like notation, where the non-terminals correspond to types. From this specification, parsers that allow the programmer to write the new data-types in whatever concrete syntax she chooses can be automatically synthesized. Furthermore, pattern matching can also be extended to use concrete syntax. Several contemporary theorem provers allow their users to extend concrete syntax of expressions [139, 106] by providing an interface to the underlying parser and pretty-printer. Visser [142] has also investigated meta-programming with concrete object-language syntax in the context of the term-rewriting language Stratego [141].

An interesting question is whether conctypes could be extended to handle object-language syntax with Haskell judgments in a type-safe way. Such an extension would have to synthesize (or otherwise allow the user to insert) appropriate equality proofs. Furthermore, how such conctypes would be typed is not immediately obvious. Pursuing this question would provide an interesting direction for future work.

9.2 Dependent Types, Type Theory and Meta-programming

9.2.1 Background

Logical frameworks were introduced by Harper, Plotkin and Honsell [53, 54] as a "formal meta-language for writing deductive systems." This work was similar to the earlier work of Martin-Löf on type theory as a foundation for mathematics. Several theorem provers have been built that are either directly based on or closely related to the LF approach: Elf[107], Coq[139],Nuprl[21].

Nordstrøm, Petersson and Smith [97] describe at length an approach to using Martin-Löf type theory as a programming language. However, as a practical programming language the pure type theory they present

is somewhat limited. Furthermore, in the Martin-Löf type theory as presented by Nordstrøm et al., there is little attention given to pragmatics such as efficiency or ease of use. There is also no particular consideration of how such programming system might relate to meta-programming.

Several programming languages have been designed to take advantage of expressive dependent type systems. Twelf[109] can be used as a logic programming language based on higher-order unification.

Cayenne[2] is a Haskell-like functional programming language with dependent types. Cayenne, makes little effort to isolate runtime computation from type-checking. Rather, it combines dependent type theory with arbitrary recursion, making the type-checking undecidable. It is argued that in practice this is not such a significant drawback. An important example of programming in Cayenne is an implementation of a tag-free interpreter [3]. Compared to the tagless interpreters presented in Chapter 2, this implementation has two distinctive features we wish to critique.

First, we note that the lack of primitive inductive types forces the rather awkward scheme of encoding typing judgments of the object language using predicates. Unlike Coq, where these predicates could be propositions without computational content, the Cayenne implementation must manipulate them at runtime. This brings us to our second point. The lack of staging makes it difficult to see what practical gains are achieved in terms of performance over a tagged interpreter.

Xi and Pfenning study a number of practical approaches to introducing dependent types into programming languages [147, 148]. Like our approach, they are concerned with designing practical programming languages where dependent types can be used to gain efficiency and expressivity. Their solution to the problems of integrating dependent types with an ML-like language is to limit the dependencies to a particular domain (decidable subset of integer arithmetic), and use constraint solving to type check their programs. They also appear to have pioneered the use of singleton types in programming languages, inspired perhaps by Hayashi [58].

The FLINT group's work on certified binaries[116] is perhaps the most closely related to the language MetaD we proposed in Chapter 2. They divide their language into a computational and non-computational, linguistically separate parts. The computational parts are programs written in one or more computational languages, while the specification language, part of which servers as a type language for the computational languages, is shared among many computational languages. The connection between computational and specification languages is achieved through the use of singleton types. Computational-language programs are annotated with proofs of various properties. These proofs are encoded in the specification language.

Each computational language in this approach must be defined separately in the meta-theory. Shao et al. present a number of such computational languages that can be used as intermediate representations in a compiler pipeline. Then, they define type (and property)-preserving translations between them. Note

that these definitions are not written in a programming (meta-)language. It is argued that in a sufficiently powerful formalism (say the calculus of constructions), such transformations could be expressed.

In these computational languages, singleton types are "hard-wired," usually only on simple types such as integers, which is a reflection, perhaps, of their intended use as relatively low-level intermediate languages in a compiler pipeline. In a heterogeneous meta-programming framework, we expect to use the inductive families facility to allow the user to define new computational languages and provide a uniform interface to singleton types along the lines described in Chapter 3.

9.2.2 Meta-programming and Dependent Types

Program Generators with Dependent Types. Sheard and Nelson [122] investigated combining a restricted form of dependent types with a two-level language. Their type system allows them to construct dependently typed program generators, but restrict such generators to functions expressible with catamorphisms. This way, termination of program generators can always be guaranteed. In this framework, both programs and their types are expressed with catamorphisms, which makes inference also possible. For more information about type inference and dependent types, one might consult Nelson's dissertation [89]. In many ways, this work resembles and anticipates that of Bjørner [11].

Certified binaries. Shao et al [116] define a general framework for writing certifying compilers. They sketch out how such a system could work by defining a number of typed intermediate languages (e.g., a lambda-calculus, a language with explicit continuations, a closure conversion language etc). Each of these languages is strongly typed in a sophisticated type system with singleton types. These type systems allow each a program in each of the intermediate languages to encode and statically certify important invariants such as bounded array indexing. A certification-preserving compiler consists in a series of transformations between these intermediate languages. Each transformation takes a well-typed program in one intermediate language and produces a well typed program in another intermediate language, so that the certified invariants present in the input programs are true in the result of the transformation. The invariants are specified in a version of the Calculus of Inductive Constructions.

It is worth noting, however, when this system is considered as a meta-programming system, that they do not give a formal meta-language in which these transformations are defined. They conjecture that such a system could formally be specified in some type theory (e.g., Coq) but it is not obvious how to do this.

9.2.3 Program Extraction as Meta-programming

Coq. Coq [139] is an interactive theorem prover based on the Calculus of Inductive Constructions [22], itself an extension of the Calculus of Constructions [23]. The original CoC system had two sorts:[3] Set, which was impredicative, and Type which was predicative, where Set:Type. Coq extended this basic formalism with a number of useful features:

1. Inductive (and also co-inductive) definitions allow the user to define new types that resemble the algebraic data-type, rather than having to work with awkward Church encodings. The types of these inductive definitions are dependent (for a discussion of inductive families, see Dybjer [34, 35]). Consider a definition for lists of a particular size, a type that is classified by Set.

   Inductive *List* [*A*:*Set*] : *nat* → *Set* :=
   Nil : (*List A O*)
   |*Cons* : (*n*:*nat*; *x*:*A*; *xs*:(*List A n*))(*List A* (*S n*)).

   Definition *oneList* : (*List Char* (1)) := (*Cons Char* (0) 'A' (*Nil Char*)).
   Definition *twoList* : (*List Char* (2)) := (*Cons Char* (1) 'B' *oneList*).
   Definition *threeList* :(*List Char* (3)) := (*Cons Char* (2) 'C' *twoList*).

   The list type has a parameter *A* (i.e., it is a type of polymorphic lists), and a natural number index indicating the list's length. Constructing such lists is arguably as easy as constructing lists in ML. Coq can also automatically derive primitive recursion combinator(s) for *List* for writing functions that analyze lists.

2. The sort *Prop* is a "twin of *Set*," [144] also impredicative, which is intended for specifying (non-computational) propositions. Types classified by *Prop* can also be defined inductively.

   As an example, consider a membership predicate over the *List* previously defined. The inductively defined predicate `Member` has one parameter, *A* : *Set* – the type of the elements of the list, and three indexes:

   (a) The element of type *A*.

   (b) A natural number indicating the length of the list.

   (c) A list in which the membership of the first index argument is asserted.

[3] Roughly speaking, morally equivalent to Martin-Löf's universes [74].

```
Inductive Member [A:Set] : A→(n:nat)(List A n) → Prop[4] :=
  MemberHead : (a:A; n:nat; rest : (List A n))
                (Member A a (S n) (Cons A n a rest))
| MemberTail : (a,b:A; n:nat; rest : (List A n))
                (Member A a n rest) → (Member A a (S n) (Cons A n b rest)).
```

We can easily build a proof, for example, that 'B' is a member of the list ['A', 'B', 'C'] by using the following proof script:

```
Lemma x1 : (Member Char 'B' (3) threeList).
Compute. Constructor. Constructor. Qed.

Print x1.
```

```
x1 =
(MemberTail Char 'B' 'C' (2) (Cons Char (1) 'B' (Cons Char (0) 'A' (Nil Char)))
        (MemberHead Char 'B' (1) (Cons Char (0) 'A' (Nil Char))))
   : (Member Char 'B' (3) threeList)
```

The most interesting feature of Coq is the extraction of programs from proofs [105]. The idea is based on the notion of Heyting interpretation of propositions, which can give a realization of an intuitionistic proof as a functional program. Several systems (theorem provers) have been inspired by this notion to provide a way of creating executable programs from logical specifications and their proofs [59, 105, 20].

The most significant feature of Coq is that it treats the twin sorts *Set* and *Prop* differently with respect to program extraction:

1. Inhabitants of types with sort *Prop* are, for the purpose of program extraction and execution, treated as comments – to be erased from the final result.

2. Inhabitants of types with sort *Set* remain in the extracted programs. However, it can be shown through realizability results [105] that dependent types can also be removed from the extracted program: given a Coq term (program), the extraction produces a computationally equivalent F_ω program.

3. Finally, the F_ω program produced by the extraction process is mapped onto a program in one of several common functional languages (Haskell, Objective CAML, Scheme).

[4] Recall from that in Coq notation, the Π types are written using parentheses. The type shown here can be written, using the more classical Π notation as: $\Pi\_ : A.\Pi n : \mathsf{nat}.\Pi\_ : (\mathsf{List}\ A\ n).\mathsf{Prop}$

Let us consider the results of extraction for the lists with length examples above. First thing to note is that the list values with length, when extracted, correspond simply to normal list, except that each `Cons` node carries the natural number index of the length of its tail. However, in the type of the list, the natural number index does not appear at all.

```
module Main where
import qualified Prelude
data List a = Nil | Cons Nat a (List a)
oneList = Cons O 'A' Nil
twoList = Cons (S O) 'B' oneList
threeList = Cons (S (S O)) 'C' twoList
```

Now, consider extracting the membership property *x1*, defined above as a proof that 'B' is a member of the list ['A', 'B', 'C']. Note that `x1` is defined as __, which should never be evaluated in the program.[5]

```
module Main where
import qualified Prelude
__ = Prelude.error "Logical or arity value used"
x1 = __
```

Let us now critique program extraction as a technique for meta-programming (in particular, its incarnation in Coq), and contrast it with solutions we propose.

1. *Pragmatic complexity of the system.* The pragmatic complexity of the system expresses itself in two different ways.

   (a) The reasonable scenario for meta-programming with Coq would require a user to first implement the critical parts of her (meta)program in Coq, formalize and prove properties about it, and finally, to use the automatic extraction to derive a CAML or Haskell program. This extracted program must then be integrated with the existing programming environment in the target language.

   This requires the programmer to be an expert in both Coq (a system not so easily mastered by an average (meta)programmer), *and* the general programming language environment (e.g., Haskell) into which the Coq-derived programs are integrated.

   (b) Developing a large program half in Coq, half in Haskell, for example, has a considerable potential of quickly turning into a software engineering nightmare.

[5] Since Haskell is lazy, so long as no-one pulls on a logical proposition value, no runtime error is raised. Coq prohibits definitions of *Set*-based values by cases over *Prop*-based ones, so the error is never raised in practice.

2. *Integration with existing programming languages and type systems.* Recall that programs extracted from Coq are F_ω programs. Mapping such F_ω programs into a typed functional language such as Haskell or OCaml is often feasible in practice, but not always [105]. In these cases, the extracted programs cannot be well-typed in the languages targeted by the extraction mechanism. The practical solution adopted by the implementor of Coq is to insert unsafe casting operations, when extracting OCaml program.

   When the target platform of program extraction is Haskell, this approach becomes problematic, since some Haskell implementations require type information produced by the type-checker in the process of compilation.

3. *Lack of programming-language features.* Many standard programming language features, such as printing and state, cannot be accessed directly in Coq. It is possible to use these features post hoc, by transforming programs extracted from Coq by hand and integrating them into larger programs written in Haskell or OCaml. This approach, however, may adversely affect the maintainability of programs.

9.2.4 Typeful Meta-programming

An interesting approach to well-typed meta-programming, anticipating both our techniques and those of Xi and Chen [146], is presented by Bjørner [11]. The meta-language is presented as a two-level λ-calculus.

Bjørner introduces a type constructor for terms that is very similar to our encoding of typing judgments in Haskell. A object language expression is represented by a special type constructor which takes as its argument a *sort*. Sorts are types of a special (non-*) *kind*, which represent the type of object-language terms. The sorts encode, in Bjørner's case, simple types. For example, type of the object-language function from integers to integers would be `Term[intsort->intsort]`). Similarly, a meta-program which optimizes or simplifies an object-program in a (object-)type-preserving way has the type:

```
simplify :: ∀v:sort. Term[v] → Term[v]
```

The meta-language is equipped with "well-typed" constructors and deconstructors for these values. The type system of the meta-language is explicitly designed to support type inference. This inference is restricted to types that are (rank-2) polymorphic in the *sorts*. The machinery that allows this is rather complicated, using a system based on higher-order semi-unification and constraint solving. An interesting example in Bjørner's meta-language is a type-preserving `map` function:

```
map :: ∀w:sort. (∀v:sort. Term[v] → Term[v]) → Term[w] → Term[w]
map f (App(M,N)) = f (App (map f M, map f N))
```

```
map f (Lam(M,N)) = f (Lam (M, map f N))
map f (Var N)    = f (Var N)
```

The morally equivalent function in Haskell, with our typing judgment representations, would look as follows:

```
map :: (∀e1 t1. Exp e1 t1→Exp e1 t1)→Exp e t→Exp e t
map f x = case x of  Var p1→f (Var p1)
                     App e1 e2→f (App (map f e1) (map f e2))
                     Abs e1 p→ f (Abs (map f  e1) p)
                     Shift e p→f (Shift (map f e) p)
```

Unlike Bjørner's system, where many interesting types of programs that manipulate typed object-language syntax can be automatically inferred, our approach requires the programmer to explicitly manipulate equality proof objects. This seems to be a consequence of the fact that the object-language is hardwired into the system. The user could not change or redefine the notion of well-typed object-language syntax. An interesting question for future work would be whether a system like Bjørner's could be automatically synthesized from a specification of the object-language type system.

9.3 Dependent Types in Haskell

Faking it. A comprehensive description of how to simulate some aspects of dependent types in Haskell was presented by Conor McBride [75]. The technique is quite similar to the one we present in Chapter 4. First, values of any first-order monomorphic type can be lifted ("faked") into the type world:

1. For each such type `T`, a Haskell class `T t` is created.
2. For each constructor $C : t_1 \to \cdots \to t_n \to t$, a data-type $C : * \to \cdots * \to *$ is created, as well as an instance placing the data-type C into the class `T`.

For example, natural numbers are defined as follows:

```
class Nat n
instance Nat Zero
instance Nat n => Nat (Succ n)

data Zero = Zero
data Succ n = Succ n
```

Second, n-ary type families (i.e., functions from n values to types) are implemented as $(n+1)$-ary multi-parameter type classes, where the $(n+1)$th parameter is functionally dependent [66] on the previous n parameters. Consider the type family, which given a natural number n, computes a type of a function with n natural number arguments.

```
nAry :: Nat -> *
nAry Z = Nat
nAry (S n) = Nat -> (nAry n)
```

This can be encoded by a multi-parameter type class:

```
class Nat n => NArry n r | n -> r where ...
instance NAry Zero Nat where ...
instance NAry n    r   => NAry (Succ n) (Nat -> r) where ...
```

Functions typed by this type family can then be defined as members of the class `NAry`.

This technique allows the type "computed" by the type family to be computed by the Horn-clause resolution machinery already present in Haskell type-checkers. McBride explains his technique by providing a number of interesting examples, such as a `zipWith` function with a variable number of arguments, and a data-type of lists whose length can be determined statically.

The main difference between our approach and that of McBride is that we have chosen not to rely on Haskell's class system to "fake" dependent types. When we lift values of first-order monomorphic types to the type level, we do so by using an inductively defined type constructor instead of a type class.

```
data Zero = Zero
data Succ n = Succ n

data IsNat n = IsNat_Zero (Equal n Zero)
             | forall n'. IsNat_Succ (IsNat n') (Equal n (Succ n'))
```

Similarly, type families become other inductively-defined data-types. In manipulating these families, we rely on the equality proofs in these data-types and a library of equality casting and manipulating functions.

This means that we cannot rely on Haskell's type checker to compute the results of type functions. We motivate our style of "faking dependent types" in Haskell by the following two points:

1. Since we have explicit equality proofs in our type families, we can use casting combinators to perform casting across the *code* type constructor, and thus move all dynamic computation related to the faking of dependent types to an earlier stage. Runtime computation incurred by McBride's encoding of

dependent types is performed by manipulating dictionaries which are not accessible to the user, and thus could not be easily used to create truly tagless interpreters.

A point related to this, noted by McBride, is that "at runtime, these programs are likely to put much a greater strain on the implementation of ad hoc polymorphism than it was ever intended to bear." [75, page 15]

2. In terms of programming style, Horn-clause notation of Haskell class definitions is not always the most intuitive way of writing functions over types. Furthermore, code for one function (which has a dependent type that is being faked) tends to be scattered among many different instances of a single class, leading to rather brittle-looking code.

Phantom Types. Hinze and Cheney [19] have recently resurrected the notion of "phantom type," first introduced by Leijen and Meijer [70]. Hinze and Cheney's phantom types are designed to address some of the problems that arise when using equality proofs to represent type-indexed data (e.g., our typing judgment `Exp`). Their main motivation is to provide a language in which polytypic programs, such as generic traversal operations, can be more easily written. This system, which can be seen as a language extension to Haskell, also bears a striking similarity to Xi's *guarded recursive datatypes* [146], although it seems to be slightly more expressive.

The main difference between phantom types and the techniques presented in Chapters 4 and 5 lies in the treatment of equality types. Instead of explicitly embedding equality types in data-type definitions, Cheney and Hinze propose a language extension which allows the programmer to state equalities between types in a data-type definitions. For example, the typing judgment for λ-calculus would be represented as follows (note that the variables not bound by the data-declarations, e.g. `t1`, are implicitly existentially quantified):

```
data Exp e t = Var                  with e = (e',t)
             | Abs  (Exp (e,t1) t2) with t = t1->t2
             | App  (Exp e (t1->t)) (Exp e t1)
```

This definition has a couple of advantages over the definitions with explicit equality proofs. First, the "smart constructors" are unnecessary to provide a useful interface for constructing `Exp`s. The system automatically assigns the "smart constructor" types to the regular constructors. Second, there is comprehensive support for de-constructing data-types with equality constraints.

The managing and propagation of equality proofs is handled automatically by the type-system. Equality proofs are never explicitly manipulated by the programmer. Instead, the type-checker uses unification to solve type equality constraints that arise from deconstructing `Exp` values. Furthermore, certain equality proof manipulation operations that cannot be implemented in Haskell, but rather have to be declared as

primitive, [6] need not be used since the built-in constraint-solver in the type checker is powerful enough infer the equalities they are used to compute.

Recasting our Haskell examples in Cheney and Hinze's language is a relatively simple exercise, as we have shown in Chapter 6. One obstacle to using Cheney and Hinze's phantom types was the lack of an implementation. This is why we developed Omega which, while still a prototype, is the first non-toy implementation of Cheney and Hinze's type system.

[6] For example the function `pairParts :: Equal (a,b) (c,d)→(Equal a c, Equal b d).`

Chapter 10

Discussion and Future Work

In the bulk of this dissertation, we have elaborated on a general framework based on programming languages, type systems, and techniques, that supports the practice of heterogeneous meta-programming. We have thoroughly explored the design space of meta-language features intended to guarantee that meta-programs maintain semantic invariants of object-language programs. In Chapter 6, we have described a functional programming language equipped with built-in support for type equality, and demonstrated its power as a meta-programming language by implementing a number of interesting examples (Chapters 6-8). In this Chapter, we summarize our findings and results, and discus directions for future work.

10.1 Thesis and Findings

Recall that the thesis of this dissertation, stated in Chapter 1.1, is that heterogeneous meta-programming can be made into a useful meta-programming paradigm that can provide some of the same benefits as the homogeneous meta-programming languages:

1. safety features (e.g., type safety of object-language interpreters, memory and separation safety in imperative object languages),
2. increased efficiency derived from the combination of semantic properties and staging,
3. type-safe object-language generation and manipulation

The first question we asked ourselves was whether it was possible to manipulate object-language programs that are not only syntactically correct, but also *type correct*? As a first step toward answering this question, we have designed a meta-language for heterogeneous meta-programming. The key property of this meta-language is that it allowed us to define the algebraic data-type representing abstract syntax in a way that preserves a notion of well-typedness of the object-language as a statically checkable invariant.

This meta-language, called MetaD, combined, roughly speaking, type system with dependent types with staging in the style of MetaML.

Next, after implementing a prototype of MetaD, we used it to write a "well-typed interpreter" (in the sense of Augustsson and Carlsson [3]) for a typed object language. This implementation exemplified a step-by-step methodology for an important class of heterogeneous meta-programming applications, which we reiterate here:

1. Start with a (strongly) typed object language. Variable binding in the language should utilize the index-based technique of de Bruijn [13]. While this can make the formal encoding of the object language somewhat awkward, it greatly simplifies the implementation.

2. Use the dependently typed inductive definition facility of MetaD to encode the typing judgments of the object-language terms as an inductive family.

3. Write a semantics for the object language. We have found that the "categorical style" semantics, an inductive mapping from the typing judgments of the object language to an arrow from the meaning of type assignments to the meaning of types, fits most naturally into our methodology. Such a semantics is then implemented as a definitional interpreter in the meta-language by providing the following:

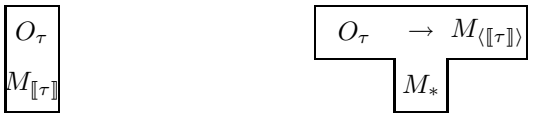
   (a) A map from the syntactic representation of object-language types to their meanings as meta-language types.

   (b) A map from the syntactic representation of object-language type assignments to the type of the runtime environments in the meta-language.

   (c) Finally, a map from the typing judgments of object-language expressions (Step 2) to "arrows" from the meaning of the associated typing assignment to the meaning of the object-language type.

4. Reformulate the interpreter (3) by adding staging annotations [117]. Accomplishing this step makes the deconstruction of typing judgments happen at the first stage, yielding a interpretive-overhead free residual program. We make an observation that such a program is free of both interpretive and tagging overhead (when the object language is a typed λ-calculus) and can thus be considered as a simple, though reasonably efficient form of compilation [38].

   Figure 10.1 illustrates the general point of this transformation by a way of "T-diagrams". The diagram (a) corresponds to an interpreter of step 3. Note that we have annotated the object-language program O, with its object-language type τ. The current step (4) accomplishes the transformation to diagram (b), where a meta-program M_* transforms the object-language program O_τ into another program in



(a) Interpreter *(b) Staged Interpreter*

O_τ	Object language program with (object) type τ
$M_{[\![\tau]\!]}$	Meta-language program of type $[\![\tau]\!]$
M_*	A tagless staged interpreter written in the meta-language.
$M_{\langle[\![\tau]\!]\rangle}$	A residual program generated by the staged interpreter.

Figure 10.1:

the meta-language, $M_{\langle[\![\tau]\!]\rangle}$, while preserving the relationship between the object-language type τ and the metalanguage type $\langle[\![\tau]\!]\rangle$.

5. Implement an associated class of meta-programs that, given a (possibly ill-typed) syntactic representation of an object-language term, constructs, if possible, a valid proof of its typing judgment that can be executed using the interpreter (defined in Step 4).

The steps (1)-(5) can be considered a paradigmatic example of heterogeneous meta-programming, consisting of an object language and an interpreter-based implementation of such a language. This kind of implementation in the meta-language we proposed has the following features:

1. The implementation of the interpreter for the object-language program ensures *statically* that only well-typed object-language programs are interpreted. If the object-language is strongly typed, the absence of runtime type errors is guaranteed in the interpreter by the type system of the meta-language.[1]

2. Adding the infrastructure for explicit staging to such an interpreter allows us to leverage the strong typing properties of the object-language to implement more efficient interpreters that do not require injecting their results into a universal domain by tagging values at runtime.

   Inspection of code generated by staging clearly reveals the absence of tags. We refer to an empirical study comparing programs generated by staged interpreters with and without tags [62]. In this study, Huang and Taha show that in practice removing tags from residual programs generated by MetaML results, on average, in twofold speedup of programs. We expect these results to hold for our implementations as well.

3. Tagless interpreters are an example of programs that analyze typed object-language programs. We have also shown how to build "parsing" functions that construct such object-language programs in a

[1] Provided, of course, that the meta-language is type safe.

type-safe way. We have implemented object-language type preserving syntax-to-syntax transformations (substitution example in Chapter 6.4), as an example of meta-programs that both analyze and construct well-typed object-language terms.

Next, we explored the question of whether the various language features present in MetaD could be harmoniously combined? To answer this question, we gave a formal definition of a small calculus that has all the ingredients of a heterogeneous meta-programming meta-language: a form of dependent types and staging. We proved that such a language is type safe with respect to an operational semantics. While this does not constitute a formal proof that the more general programming language MetaD is type safe as well, it represents a good prima facie evidence that the main ingredients of MetaD *can* be safely and orthogonally combined in a single language. A full formalization of the part of MetaD that contains dependent families is left as a question for future work.

The next question we asked ourselves was whether the full expressive power of the dependent types used in MetaD is really necessary for the meta-programming paradigm outlined above. Can the typing judgments of object languages be encoded using something more akin to data-types in functional programming languages than the inductive type families of the calculus of constructions?

Here, we demonstrated how such encodings (as well as definition of well-typed, tagless interpreters, type-checkers, and other meta-programs) can be accomplished using a technique for encoding type equality in Haskell-like languages. Thus, we have successfully recast the object-language implementation methodology (full steps 1-5) in Haskell.

The advantage of this approach is primarily in obtaining a more practical heterogeneous meta-programming platform. However, we discovered significant practical drawbacks of this approach as well: explicit manipulation of type equality encodings in Haskell can be both cumbersome and inefficient.

What was needed is a meta-programming language that allowed the programmer to use type equality in his encodings of abstract syntax, but automated much of the tedium of type equality manipulation. This is why we designed and implemented the functional language Omega, the first implementation of a functional language with automatic type equality manipulations.

In designing Omega, we started with a functional language similar to Haskell. We modified its type system to automatically keep track of type equalities. The most important new language feature we added was a generalized algebraic data-type definition facility which allowed to programmer to specify equalities between types that must hold for each constructed element of the data-type. We implemented a type checker that automatically proves and propagates these type equalities through the meta-program. With Omega, we were able to implement all our Haskell examples in a cleaner, simpler style. We evaluated Omega's usefulness as a meta-language on an expanded set of meta-programming examples.

Examples. Another modality of support, to which our thesis lends itself naturally, is by detailed examples which showcase a set of techniques which, taken together, make up a practical idiom for heterogeneous meta-programming. Also, the examples have the nature of a tutorial – the goal is to teach readers interested in heterogeneous meta-programming how to implement an important class of heterogeneous meta-programs by describing step by step implementations.

Evaluation. In Chapter 1 we have outlined a set of criteria that a usable heterogeneous meta-programming language should fulfill. We review these criteria and comment on how the work presented in the rest of this dissertation addresses each of them.

1. *Is it possible to define and manipulate* different *object languages?* Yes. In Chapter 3 we have demonstrated how an example object-language based on the simply typed λ-calculus can be encoded as an inductive family in MetaD. In Chapter 5, we have started with an encoding of the same λ-calculus based language. Then, we extended the object language, non-trivially, with pattern matching to demonstrate how a wider variety of object-language features can be treated with our technique. Then, we have shown (Chapters 6-8), how an even more interesting set of object-language type systems can be encoded, this time using Omega: $\lambda^{\Box}$, $\lambda^{\bigcirc}$ and the calculus of explicit substitutions.

2. *Is it possible to statically enforce important object-language properties such as typing and scoping?* Yes. In Chapter 3, we show how to produce such encodings in dependently typed language MetaD by using dependently typed inductive families.[2]

   In Chapter 5 we have described a technique that allows us to do this in Haskell. While Haskell, as a meta-language, does not guarantee the soundness of the logical predicates that encode object-language properties, we discuss how this problem can be handled in practice. Finally, in Chapter 6 we demonstrate how such invariants are enforced in Omega.

3. *Can we write efficient meta-programs?* Yes. A classical way of achieving efficiency in interpreters (and other meta-programs) is by applying staging techniques to them [128, 117]. Having described a way of encoding object-language abstract syntax that, as McBride put it, "allowing us to equip data-structures [and abstract syntax] with built-in invariants" [75], is there any useful that staging can play in the larger scheme of things?

   We have been able to demonstrate how we can derive tangible benefits from using both typeful representations of object-language syntax *and* staging. As a demonstration, we develop implementations of tagless staged interpreters, thus providing a plausible solution to the problem posed by Taha in

[2] For comparison the reader might peruse Appendix A for a comparative development in Coq.

the context of MetaML [128, page 151[3]]. In MetaD and Haskell staging plays an important role in producing efficient tagless interpreters, since manipulating typing judgments/equality proofs incurs some runtime overhead that can be removed using staging.

In Omega, we can write truly tagless interpreters (removing all tagging overhead) without the need for staging, since the Omega type system performs type equality proof manipulation statically, at type-checking time. Staging can still be used in Omega to remove interpretive overhead from tagless interpreters.

4. *How easy is it to integrate it into functional programming in general?* The answer is a qualified 'yes.' This is a pragmatic question that we have explored in the second part of the dissertation (starting with Chapter 4). A plausible criticism of MetaD (and, to some extend, of Coq) is that it is a "toy" implementation that one cannot easily integrate with "real" functional languages. By describing a way of encoding well-typed syntactic judgments of object-language programs, we have argued that heterogeneous meta-programming can be made available to the "broad masses" of Haskell programmers.

   Most importantly, we have shown how type equality can be incorporated into a practical programming language (Omega). Built-in type equality provides the meta-programmer a generalization of traditional algebraic data-types that we demonstrate to be as useful as MetaD inductive families in practical meta-programming. At the same time, to a programmer already familiar with functional programming and algebraic data-types, the mechanisms in Omega present a significantly less steep learning curve than dependent types.

10.2 Future Work

Finally, we conclude our exposition by outlining several areas for future work on heterogeneous meta-programming.

Faking dependent types. In this dissertation we have presented a number of examples of encoding typing judgments of various object languages in Haskell and Omega. We may also observe that this technique is a instance (in the context of heterogeneous meta-programming) of a general technique for "faking" dependent type-like behavior in functional languages with a sufficiently expressive type system. The question that arises, then, is "how complete[4] is this 'faking' technique?" We do not provide a formal, rigorous answer

[3] *"Are there practical, sound type systems for expressing tag-less reifiers?"*

[4] A similar question about soundness has a rather facile negative answer, since in Haskell all types are inhabited. However, it is not unreasonable to argue that with a modicum of self-discipline, this question need not adversely affect the programmer in practice.

to this question; rather, we concentrate on exploring, through examples, a class of problems where the technique is sufficient for interesting applications. However, there are some negative observations about the power of the technique that we can formulate.

We recall that in the object-language typing judgments that we have defined, the indexes (representing object-language types and type assignments in the types of object-language judgments) have all been first order data-types. This property has made it easy to encode (simulate) the values of those indexes *at the type level* in the meta-language. What if, however, we wish to encode higher-order values at the type level? Two related problems present themselves:

1. Presumably, we would like to represent functions, say of type $\texttt{t1} \rightarrow \texttt{t2}$ by *type constructors* of kind $* \longrightarrow *$, with the appropriate side conditions that the argument and the result of such a type constructor is only used on type-level representations of `t1` to yield type-level representations of `t2`. However, type constructors in Haskell are not really functions on types – they are syntactically restricted to an applicative form of already-defined type constructors. No β or similar computational rules apply to them.

   When we simulate type-family computations (i.e., computing a type based on a value simulated at type level), we rely on Haskell (or Omega's) type checker's implementation of unification to perform the actual work of computation. Since type checkers for functional languages cannot be relied on to perform sophisticated computations if they are to preserve type inference, we often have to help it along by supplying equality proofs and various casting operations in our programs.

   As we have noted already, the lack of real functions at type level also means that we cannot perform evaluation on such type constructors either.[5]

2. Another technique that makes faking dependent types in Haskell/Omega usable is the ability to have runtime representations of values that are encoded at type-level. These runtime representations can be compared (not surprisingly, at runtime) to yield equality proofs, which, in turn, can be used to cast between such representations. For example, this is a technique heavily relied on by the function `typeCheck` in Chapter 5.

   However, it is not quite clear what a runtime representation of a *type constructor* would look like, especially considering the fact that only values classified by types of the kind * can exist at runtime.

Let us further illustrate the problems above by an example. Suppose we have defined a type of lists in

[5] This is not completely true, since perhaps we could encode S and K combinators at type level, and perhaps create inductive judgments that would allow us to drive a form of reduction of such combinators "from below", by designing carefully constructed data-types at runtime. However, this is not a practical solution.

Omega, so that the length of the list is encoded in its type:

```
-- kind Nat = Z | S Nat
data List a n = Nil where n = Z
              | ∀ m. Cons a (List a m) where n = S m
```

The type of the function that appends two list can be most naturally expressed so that the length of the list it returns is the sum of the lengths of its two arguments:

```
append :: List a m → List a n → List a m+n
```

However, neither Omega nor Haskell allow us to write such a type since we cannot define addition as a function at the level of types. The current solution is to encode addition as a relation between natural number at the level of types:

```
append :: Plus m n q → List a n → List a m → List a q
```

This style is both unnecessarly complex (since it introduces confusing auxiliary judgments), and inefficient (since we must construct and manipulate the proof of `Plus` at runtime). The question then is whether Omega's type system can be suitably extended to make it possible to address them, perhaps by allowing a restricted form of functional values at type level.

Logical framework in Haskell. The question then becomes whether Omega's type system can be suitably extended to make it possible to address them, perhaps by allowing a restricted form of functional values at type level.

Logical framework in Haskell. Related to the previous question is whether we could embed a sufficiently expressive logic into Haskell/Omegaby various 'faking' techniques? In other words, can we fake our way into a logic powerful enough to allow us to write non-trivial, predicate based, specifications (and proofs of those specifications) of executable Haskell/Omegaprograms?

The details of this question remain both tantalizing and elusive.

Object-language binding constructs. The use of de Bruijn indices to represent binders in object-languages has been used throughout this dissertation. We are well-aware that this technique is both awkward and error prone.[6] When it comes to representing syntax with binders, at least until recently, one could feel

[6] We note, in passing, that de Bruijn indices seem to be less error-prone in typeful syntax representations, since static type-checking can catch a lot of "off-by-one errors" that tend to creep into programs manipulating de Bruijn-style terms.

justified in paraphrasing a famous apothegm of Churchill's: de Bruijn's is the worst style of representing binding constructs, except for all those others that have been tried. However, there are some glimmers of hope.

In the context of mechanical theorem proving, McKinna and Pollack [76] present certain formalizations of the λ-calculus and type theory (PTSs) without resorting to de-Bruijn representation of terms. Their technique, however, seems much more applicable to theorem proving than to meta-programming.

We have already discussed Nanevski's adaptation of Pitts and Gabbay's nominal logic to meta-programming in $\lambda^{\Box}$. In particular, Nanevski introduces a type constructor $(A \not\rightarrow B)$ for "binding a (fresh) object variable of type A in an object-program of type B." In the future, we plan to explore how such a construct can be adapted to representing typeful object language syntax. This direction seems to show most promise.

Related to this concern is the question we raised in Section 9.1.2 of whether we can integrate some support for pretty-printing and parsing that would allow us to interface with object-language programs using concrete syntax.

Semantic properties of object programs. In this dissertation, we have concentrated on representing one class of properties of object-language syntax, namely, type correctness. This seems reasonable in the context of heterogeneous meta-programming, since manipulating object-language syntax that is well-typed is quite useful. However, an interesting question for future work is whether there are *other* interesting properties that could be easily encoded and integrated into our heterogeneous meta-programming framework. One such example would be to encode safety properties of programs (e.g., proof of array bounds checking, division by zero, and so on). Another interesting question is whether we can have object-language representation with multiple *kinds* of properties, integrated into a single heterogeneous meta-programming framework in a modular way.

Appendix A

Tagless Interpreters in Coq

A.1 Introduction

Coq [139] is an interactive theorem prover based on the Calculus of Inductive Constructions [22]. Since the underlying logic of Coq is a constructive type theory, the theorem prover has been designed to support *extraction* [104] of programs from proofs and definitions. Such a system seems at first to be at least a good candidate for the kind of meta-language that we are seeking in which to implement object language manipulations that preserve semantic properties. Tagless interpreters are an excellent example of such a manipulation, and its development using Coq is indeed possible to a large extent.

```
Inductive nat : Set := O : nat | S : nat→nat.
Inductive 𝕋 : Set := N : 𝕋 | ArrT : 𝕋 → 𝕋 → 𝕋.
Inductive 𝔼 : Set := Const : nat → 𝔼 | Var : nat → 𝔼 | Abs : 𝕋 → 𝔼 → 𝔼 | App : 𝔼 → 𝔼 → 𝔼.
Inductive 𝔾 : Set := Empty : 𝔾 | Ext : 𝔾 → 𝕋 → 𝔾.

Inductive HasTypeVar : 𝔾 → nat → 𝕋 → Prop :=
  HasTypeVar_Zero : (Γ:𝔾;t:𝕋)(HasTypeVar (Ext Γ t) O t)
| HasTypeVar_Weak : (Γ:𝔾;n:nat; t,t' : 𝕋)(HasTypeVar Γ n t)→ (HasTypeVar (Ext Γ t') (S n) t).

Inductive HasType : 𝔾 → 𝔼 → 𝕋 → Prop :=
  HasType_Const : (Γ:𝔾;n:nat)(HasType Γ (Const n) N)
| HasType_Var : (Γ:𝔾;n:nat;t:𝕋) (HasTypeVar Γ n t) → (HasType Γ (Var n) t)
| HasType_Abs : (Γ:𝔾;t1,t2:𝕋;e:𝔼) (HasType (Ext Γ t1) e t2) → (HasType Γ (Abs t1 e) (ArrT t1 t2))
| HasType_App : (Γ:𝔾;t1,t2:𝕋;e1,e2:𝔼) (HasType Γ e1 (ArrT t1 t2)) → (HasType Γ e2 t1) →
                (HasType Γ (App e1 e2) t2).
```

Figure A.1: Syntactic Definitions for L_0 in Coq.

For pedagogical reasons, then, we shall first develop a tagless interpreter using the Coq system. We will introduce Coq syntax and operations as we go along. The reader is referred to the excellent tutorial by Gimenez [44] for more systematic instruction.

The Figure A.1 is a Coq script defining the syntax and the type system of L_0. This script consists of a

series of *inductive* definitions.

Let us briefly examine the syntax of one of these definitions:

Inductive $\mathbb{T}$: *Set* := *N* : $\mathbb{T}$ | *ArrT* : $\mathbb{T} \to \mathbb{T} \to \mathbb{T}$.

The inductive family can most easily be thought of as a data-type in traditional functional languages. The above definition introduces a new type $\mathbb{T}$. This new type is itself given the type *Set* (more on this later). Following the assignment sign (:=), we list a number of constructors, where each constructor is given its full type. Naturally, the result type of each constructor must be $\mathbb{T}$. After accepting this definition, Coq allows the user to use $\mathbb{T}$ very much as one does a data-type in Haskell or ML: expressions of type $\mathbb{T}$ can be examined with *case* expressions, and recursive functions (provided that they are indeed primitive-recursive, i.e., that they terminate) can be defined over them.

Inductive definitions go beyond data-types in the sense that they allow the inductive families to be dependently typed. The inductive families *HasTypeVar* and *HasType*, in Figure A.1, are an example of such dependent typing.

A.1.1 A Brutal Introduction to Coq Syntax

Before we dissect these definitions, let us review the syntax of Coq. In addition to traditional function type former $\tau_1 \to \tau_2$, Coq supports the dependent function space $\Pi x : \tau_1.\tau_2$. In Coq syntax, this is written by prepending parentheses which bind a variable whose scope extends to the right: (*x*:*T1*)*T2*. Multiple nested Π-types, $\Pi x_1 : \tau_1.\Pi x_2 : \tau_2 \ldots \tau_n$ can be combined in the syntax as (*x1*:*T1*; *x2*:*T2*; ...)*Tn*. Further syntactic sugar is provided when Π-abstracting over multiple variables of the same type: $\Pi x_1 : \tau.\Pi x_2 : \tau.\tau'$ can be written as the Coq type (*x1*,*x2*:*T*)*T'*. As is usual with dependent types, the function type *T1*→*T2* is just syntactic sugar for $(\Pi\_ : T_1.T_2)$ (in Coq notation, (_:*T1*)*T2*).

Function abstraction $\lambda x : \tau.e$ is written in Coq the same as the Π-abstraction, except that square brackets are used instead of parentheses: [*x*:*T1*]*e* is a function that takes an argument *x* (of type *T1*) and whose body is the expression *e*. In all binding constructs that require typing annotation (e.g., [x:*T1*]*T2*) the user can omit the type of the variable provided that the type can be inferred from context by placing a question mark instead of a term (e.g., [x:?]*T2*). If the inference is impossible, the system complains.

It is also worth noting that, contrary to common practice in functional programming, application (*e1 e2*) has lower priority in Coq than various binding constructs. Thus, the expression [*x*:*T*]*x y* is fully parenthesized as (([*x*:*T*]*x*) *y*.

Returning to inductive definitions, let us consider the definition of the inductive family *HasTypeVar*. The inductive family *HasTypeVar* corresponds to the auxiliary typing judgment VAR $\Gamma \vdash n : \tau$ from Figure 2.2. It is defined as follows:

Inductive *HasTypeVar* : $\mathbb{G} \rightarrow$ *nat* $\rightarrow \mathbb{T} \rightarrow$ *Prop* :=

HasTypeVar_Zero : (Γ:$\mathbb{G}$;*t*:$\mathbb{T}$)(*HasTypeVar* (*Ext* Γ *t*) *O t*)

| *HasTypeVar_Weak* : (Γ:$\mathbb{G}$;*n*:*nat*; *t*,*t'* : $\mathbb{T}$)(*HasTypeVar* Γ *n t*)→ (*HasTypeVar* (*Ext* Γ *t'*) (*S n*) *t*).

A couple of points are worth noting:

- As in the definition of the inductive family $\mathbb{T}$, the type family *HasTypeVar* itself must first be given a type. Rather than just *Set*, the type *HasTypeVar* is a function taking three arguments (sometimes called indexes): a type assignment $\mathbb{G}$, a natural number *nat*, and a L_0 type $\mathbb{T}$, and returning the *sort* *Prop*. In a way, this is analogous to a Haskell notion of *type constructor*, except that whereas Haskell type constructors are functions from types to types (in Coq one would write them as *Prop* → *Prop*), Coq type families are functions from *values* to types.

  One can think of *sorts* as special types that classify other types. The sort *Prop* is a type of logical propositions/formulas. The sorts *Set* and *Prop* are similar to the notion of *kind* * in Haskell, except that Coq divides the kind of types into two distinct parts: one reserved for programs and values (*Set*) and the other reserved for logical propositions (*Prop*). Logically, this distinction is not strictly necessary: *Set* by itself would be sufficient. Indeed, most dependently typed languages unify *Set* and *Prop* into one single sort (e.g., Cayenne [2]). However, as we will see later, *Prop* and *Set* can be given different "operational" properties if we treat Coq definitions as programs: *Set* types become types of runtime values (integers, functions and so on), while *Prop* types become mere logical properties of those values which are used at type-checking but are discarded from the runtime computation.

- The values of the inductive family *HasTypeVar* can be built up using the two constructors, *HasTypeVar_Zero* and *HasTypeVar_Weak*. The types of these constructors merit a closer examination:

  1. The constructor *HasTypeVar_Zero* is the base case of the typing judgment on variables. It corresponds to the (Var-Base) rule from Figure 2.2:

     $$\frac{}{\mathsf{VAR}\ \Gamma, t \vdash 0 : t}\text{(Var-Base)}$$

     It has the dependent type (Γ:$\mathbb{G}$;*t*:$\mathbb{T}$)(*HasTypeVar* (*Ext* Γ *t*) *O t*). This means that it is a dependently typed function which takes two arguments, Γ of type $\mathbb{G}$ and *t* of type $\mathbb{T}$, and returns a value of type (*HasTypeVar* (*Ext* Γ *t*) *O t*).

2. The constructor *HasTypeVar_Weak* is the weakening (inductive) case of the typing judgment on variables. It corresponds to the Var-Weak rule from Figure 2.2:

$$\frac{\mathsf{VAR}\ \Gamma \vdash n : t}{\mathsf{VAR}\ \Gamma, t' \vdash (n+1) : t}(\text{Var-Weak})$$

It also has a dependent type:

(Γ:𝔾;*n*:*nat*; *t*,*t'* : 𝕋)(*HasTypeVar* Γ *n t*)→ (*HasTypeVar* (*Ext* Γ *t'*) (*S n*) *t*).

Again, the constructor itself is a dependently typed function. Its first argument is the type assignment Γ. Its second argument is a natural number *n*. Its next two arguments are two L_0 types *t* and *t'*. Finally, its last argument is a typing judgment of type (*HasTypeVar* Γ *n t*). Given all these arguments, its result is of type (*HasTypeVar* (*Ext* Γ *t'*) (*S n*) *t*).

A closer examination reveals that this type corresponds exactly to the inductive definition of judgments in Figure 2.2: the last argument to the constructor corresponds to the antecedent of the rule. The result of the type corresponds to the rule consequent. The arguments preceding the antecedent simply serve to "close" the free variables in the judgment, which in Figure 2.2 are implicitly universally quantified.

Now, having defined L_0 well-typedness judgments as Coq inductive families, they can be treated as provable Coq propositions.

A.1.2 A Brutal Introduction to Theorem Proving in Coq

But first, we shall briefly digress here to review the process of theorem proving in Coq. Due to the Curry-Howard isomorphism[61, 97], to prove a proposition $\mathcal{P}$ in Coq, all one has to do is to construct an *inhabitant* of the type that corresponds to $\mathcal{P}$.[1] Usually, propositions are types whose sort is *Prop*, although the theorem prover is also capable of interactively constructing inhabitants of types with sort *Set* as well.

As an example, consider the judgment $(\mathsf{EXP}\ \langle\rangle, N \vdash \lambda(N \to N).(\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1) : (N \to N) \to N)$, i.e., that the L_0 expression $\lambda N \to N.\,(\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1)$ has type $(N \to N) \to N$ under the type assignment $\langle\rangle, N$. First, we write down the appropriate Coq type that corresponds to this proposition: (*HasType* (*Ext Empty N*) (*Abs* (*ArrT N N*) (*App* (*Var* (0)) (*Var* (1)))) (*ArrT* (*ArrT N N*) *N*)).

Next, we issue the command Theorem, and give a name under which the inhabitant of this type will be known to the system (*example1*):

```
Theorem example1 : (HasType (Ext Empty N)
                            (Abs (ArrT N N) (App (Var (0)) (Var (1))))
```

[1] In other words, find an expression e whose type is $\mathcal{P}$.

(*ArrT* (*ArrT N N*) *N*)) .

After this command is issued, Coq goes into the interactive theorem proving mode. It prints the type of *example1*, declared above, as a goal (below the line) and prompts the user for next command:

Coq output

```
1 subgoal

  ============================
   (HasType (Ext Empty N) (Abs (ArrT N N) (App (Var (0)) (Var (1))))
     (ArrT (ArrT N N) N))
```

Now, we issue the command *Proof* to begin proving the theorem. The first tactic we chose to use is the tactic *EApply* (we also sometimes use a closely related tactic called *Apply* – the reader can assume them to be basically equivalent). This tactic takes an argument expression *e*. The theorem prover first computes the type of *e*. If it is an arrow type, it tries to unify its result type with the type of the current goal. If the unification succeeds, the current goal is replaced by the types of the arguments to the function. If the type of *e* is not a function and the unification with the current goal succeeds, the goal is eliminated. The argument we give to *EApply* is the constructor *HasType_Abs*:

EApply HasType_Abs.

Having succeeded in the previous tactic, the theorem prover prints out a new goal:

Coq output

```
1 subgoal

  ============================
   (HasType (Ext (Ext Empty N) (ArrT N N)) (App (Var (0)) (Var (1))) N)
```

Since the expression for which we are constructing the proof now is an application, it is a good idea to try to apply the constructor *HasType_App*:

EApply HasType_App.

Again, the tactic succeeds. Now the system introduces two new goals (one for each of the two *HasType* arguments to *HasType_App*), and prints:

Coq output

```
2 subgoals

  ============================
   (HasType (Ext (Ext Empty N) (ArrT N N)) (Var (0)) (ArrT ?3 N))

subgoal 2 is:
 (HasType (Ext (Ext Empty N) (ArrT N N)) (Var (1)) ?3)
```

We could continue to use *EAppy* with *HasType* constructors, but Coq has much more sophisticated tactics that can figure out automatically what constructors to apply. One such tactic is called *Constructor*:

Constructor.

Now, a new subgoal is created instead of the first goal. Notice that it has used the constructor *HasType_Var* and the new goal is a variable judgment:

Coq output

```
2 subgoals

  ============================
   (HasTypeVar (Ext (Ext Empty N) (ArrT N N)) (0) (ArrT ?3 N))

subgoal 2 is:
 (HasType (Ext (Ext Empty N) (ArrT N N)) (Var (1)) ?3)
```

We quickly dispense with this subgoal by instructing the theorem prover to repeatedly apply the *Constructor* tactic until it proves the current goal:

Repeat Constructor.

Now we are left with only one goal (former subgoal 2):

Coq output

```
1 subgoal

  ============================
   (HasType (Ext (Ext Empty N) (ArrT N N)) (Var (1)) N)
```

Again, we dispense with it using the *EAuto* tactic (which combines *Constructor* with other automatic reasoners):

EAuto.

And we are done! The system prints:

Coq output

```
Subtree proved!
```

This indicates that all the subgoals have been discharged and the proof is completed. We issue the one final command *Qed*, to instruct the prover to accept the definition of *example1* we have just interactively constructed:

Coq output

```
EApply HasType_Abs.
EApply HasType_App.
Constructor.
Repeat Constructor.

EAuto.

example1 is defined
```

To review, the above theorem is defined using a proof script which consists of a series of commands (tactics) given to the interactive theorem prover between commands *Proof* and *Qed*. Due to the type-theoretic nature of Coq, the proof of the theorem constructed above can also be viewed as a program that the theorem prover constructs interactively. Thus, requesting the system to print the value of the variable *example1* yields the following response:

Print *example1*.

Coq output

```
example1 =
(HasType_Abs (Ext Empty N) (ArrT N N) N (App (Var (0)) (Var (1)))
  (HasType_App (Ext (Ext Empty N) (ArrT N N)) N N (Var (0)) (Var (1))
    (HasType_Var (Ext (Ext Empty N) (ArrT N N)) (0) (ArrT N N)
      (HasTypeVar_Zero (Ext Empty N) (ArrT N N)))
    (HasType_Var (Ext (Ext Empty N) (ArrT N N)) (1) N
      (HasTypeVar_Weak (Ext Empty N) (0) N (ArrT N N)
        (HasTypeVar_Zero Empty N)))))
     : (HasType (Ext Empty N)
         (Abs (ArrT N N) (App (Var (0)) (Var (1)))) (ArrT (ArrT N N) N))
```

Further Tactic Examples. To round off this little tutorial, we shall give another example of interactive theorem proving to introduce the user to the tactics that will be used later on in this chapter. First, one should recall that the interactive prover is not limited to proving propositions, but can be used to construct the inhabitants of any Coq type. We shall thus consider defining an inhabitant of the type (*m*,*n*:*nat*)*nat*, in particular, the addition function. Since this type is not a *Prop*, we shall use the keyword *Definition* instead of *Theorem* to enter into the interactive mode:

Definition *plus* : (*m*,*n*:*nat*)*nat*. *Proof*.

The first thing that happens when entering the interactive mode is that Coq prints the type of the goal we are trying to define:

Coq output

```
1 subgoal

  ============================
   nat->nat->nat
```

We will opt to define this function by recursion on its first argument *m*. We issue the command *Fix* 1, and the theorem prover responds with:

Coq output

```
1 subgoal

  plus : nat->nat->nat
```

```
  ============================
   nat->nat->nat
```

We can see that we have acquired a new *assumption*, named *plus* which has the same type as the value we are trying to define. This assumption corresponds to a recursive call to the function *plus* itself.

Next, since we are trying to prove an implication (->), we can instruct the prover to use the implication introduction rule as far as possible. The tactic *Intros* does just this.

Coq output

```
1 subgoal

  plus : nat->nat->nat
  m : nat
  n : nat
  ============================
   nat
```

Now, we have two more *assumptions*, named *m* and *n*, of type *nat* and are trying to show an inhabitant of *nat*. Any *nat* would logically do, but we want to define a particular *nat* that is the sum of *m* and *n*. This can be best accomplished by examining the cases over one of the assumptions, say,*m*. We issue the following command:

NewDestruct m.

Now, the prover has split the proof into two subgoals

1. The first case is when *m* is zero:

   Coq output

   ```
   2 subgoals

     plus : nat->nat->nat
     n : nat
     ============================
      nat

   subgoal 2 is:
    nat
   ```

   But, if *m* is zero, then that sum of *m* and *n* is just *n*, and we can issue the appropriate command:
   Apply n.

2. Thus, the first goal is discharged. Now, for the inductive case where *m* is of the form *S n0*, for some natural number *n0*:

   Coq output

   ```
   1 subgoal
   ```

```
plus : nat->nat->nat
n0 : nat
n : nat
============================
 nat
```

Well, we know that since *m*=*S n0*, $m + n = S(n0 + n)$, we can immediately provide the natural we want:

Apply(S (*plus n0 n*))

This discharges all subgoals, and we exit the interactive mode by the command *Defined*, which is analogous to *Qed* (there *is* a slight, but for our purposes unimportant difference in how the theorem prover keeps track of values depending on which of the two commands is used).

We can also instruct Coq to print the definition of *plus*:
Print *plus*:

Coq output

```
plus =
Fix plus
  {plus [m:nat] : nat->nat :=
     [n:nat]Cases m of
              (0) => n
            | ((S n0)) => (S (plus n0 n))
            end}
     : nat->nat->nat
```

A.1.3 Semantics of L_0 in Coq

The next step is to define the semantics of the language L_0. As we have seen in Section 2.2.1, the semantics is defined inductively over the well-typedness judgments. In our Coq implementation, the meaning of L_0 types is a function that maps $\mathbb{T}$s into *Set*. Similarly, type assignments are also mapped to *Set*, i.e., to a nested product of the meaning of individual types in the type assignments.

The semantic functions *evalT* and *evalTS* are defined below by recursion on L_0 types and type assignments, respectively. This form of definition is quite similar to programs in Haskell or ML, with the exception that Coq attempts to prove that the recursively defined function always terminates by examining a particular argument (in this case the type or the type assignment) and ensuring that it is structurally smaller at every recursive call:

```
Fixpoint evalT [T:𝕋] : Set :=
  Cases T of
      N ⇒ nat
    | (ArrT t t0) ⇒ (evalT t)→(evalT t0)
  end.
```

```
Fixpoint evalTS [Γ : 𝔾] : Set :=
Cases Γ of
      Empty ⇒ unit
    | (Ext Γ′ t) ⇒ ((evalTS Γ′) × (evalT t))
end.
```

Alternatively, we can make the appropriate definitions using Coq's interactive theorem proving facility, where we use tactic to generate the code for the functions we wish to define. The convenience of this approach is that arguments available to a definition are shown as premises, while the types whose values we are trying to construct are shown as current goals. The proof environment makes sure that all the cases are addressed and that only well typed programs are constructed. Furthermore, certain Coq tactics[32] can be used as powerful program generation tools. After the definition is complete, the user can easily inspect the source of the function she has interactively defined. Thus, the semantics of types and type assignments can be defined by the following proof script:

```
Definition evalT : 𝕋 → Set. Proof. Induction 1. EApply nat. Intros. EApply (X→X0). Defined.
Definition evalTS : 𝔾 → Set. Proof. Induction 1. EApply unit. Intros. EApply (X×(evalT t0)).
Defined.
```

Syntactic notations. Another useful facility that Coq provides is to define syntactic shortcuts which allow the user a rather flexible way of extending the syntax of her programs. Prefix, infix and mixfix operators can easily be declared. For example, the following definitions allow us to use the more convenient notation $\mathcal{T}[\![t]\!]$ instead of *(evalT t)*:

```
Notation " 𝒯 ⟦ t ⟧ " := (evalT t).
Notation " 𝒯𝒮 ⟦ ts ⟧ " := (evalTS ts).
```

The syntax for these definitions is rather intuitive: concrete syntax appears on the left of the assignment sign (:=) surrounded by quotes. The corresponding Coq expression is written on the right – identifiers mentioned in both are considered as variables ranging over syntactic expressions (variable *t* above). Upon accepting a syntactic notation definition, the Coq system automatically generates parsing and pretty-printing functionality and the newly defined notations can be freely mixed with other Coq syntax.

Syntactic notations can be used to make our definition of typing judgments syntactically identical with the definitions in Figure 2.2. First, we declare notations for types, expressions and type assignments:[2]

```
Notation "t1 ⟶ t2" := (ArrT t1 t2) (at level 65, right associativity).
Notation " λ t . e " := (Abs t e) (at level 40, left associativity).
```

[2]Note that this system is not perfect, and instead of application just being $e_1\ e_2$ we had to use the infix symbol @ lest the parser confuse application in L_0 with application of Coq.

Notation " *A* @ *B* " := (*App A B*) (*at level* 50, *left associativity*).

Notation " $\langle\rangle$ " := *Empty*.

Notation "*G* ;; *t*" := (*Ext G t*) (*at level* 60, *left associativity*).

Finally, we can define a more traditional mixfix notation for typing judgments:

Notation " '*VAR*' $\Gamma \vdash n : t$ " := (*HasTypeVar* Γ *n t*).

Notation " '*EXP*' $\Gamma \vdash e : t$ " := (*HasType* Γ *e t*).

Resuming with the semantics of L_0, the next step is to define the auxiliary function *lookup*, which implements the semantics of variable look-up: it takes as its argument a type assignment, a natural number index of the variables, and returns a function from the meaning of the type assignment to the meaning of the type of the variable:

Definition *lookUp* : (Γ:$\mathbb{G}$)(*n*:*nat*)(*t*:$\mathbb{T}$)(*VAR* $\Gamma \vdash n : t$)→($\mathcal{TS}$ $[\![\Gamma]\!]$) → ($\mathcal{T}$ $[\![t]\!]$).

We shall define this function interactively, with tactics. The function is defined by recursion on the second argument, the natural number index of the variable. We use the tactic Fix 2, which gives us access to the recursive call to *lookUp*. This gives an assumption

lookUp : (Γ:$\mathbb{G}$)(*n*:*nat*)(*t*:$\mathbb{T}$)(*VAR* $\Gamma \vdash n : t$)→($\mathcal{TS}$ $[\![\Gamma]\!]$) → ($\mathcal{T}$ $[\![t]\!]$).

Next, we recall that what the type we are trying to prove is a (dependent) function type. In general, to prove the proposition (*x*:*P*)*Q*, we have to prove *Q* under the assumption *x*:*P*. In Coq, we use the tactic *Intros* to do this, and obtain the following assumptions:

Γ : $\mathbb{G}$

n : *nat*

t : $\mathbb{T}$

H : (VAR $\Gamma \vdash n : t$)

H0 : $\mathcal{TS}[\![\Gamma]\!]$

The new goal becomes $\mathcal{T}[\![t]\!]$.

The definition now proceeds by case analysis on the type assignment Γ: *NewDestruct* Γ.

Now, there are two different cases for the variable Γ:

1. Case Γ = Empty. If Γ is empty, then the assumption *H* has the type Var Empty $\vdash$ *n* : *t*. If we examine the inductive definition of the variable judgments, we notice that there is no derivation such that under the empty environment some variable index has a type, i.e., we cannot project from an empty environment.

   This means that one of our assumptions, *H*, is false, and logically, from falsity we can prove any goal.

In Coq, we shall prove this case by using the *Absurd* tactic, which takes a proposition P proves an arbitrary proposition Q, provided that both not P and P follow from the current assumptions. The formula for P we use is simply $(\mathsf{Var}\ \mathsf{Empty} \vdash n : t)$. The formula Q is, of course, the original goal $\mathcal{T}[\![t]\!]$.

Absurd (*VAR Empty*⊢*n*:*t*).

Now, the goal $\neg(\mathsf{Var}\ \mathsf{Empty} \vdash n : t)$ follows by examining the types of the constructors for *HasTypeVar* and determining that there is no derivation with an empty type environment. Coq has a tactic automatically does this: *Inversion H*.

Next, the goal $(\mathsf{Var}\ \mathsf{Empty} \vdash n : t)$ follows trivially from the assumptions: *Trivial*.

Now the initial goal $\mathcal{TS}[\![t]\!]$ has been proved.

2. Case $\Gamma = \Gamma', t1$. For the second case, the original assumptions are rewritten with respect the new value of Γ:

   $\Gamma' : \mathbb{G}$

   $t1 : \mathbb{T}$

   $n : nat$

   $t : \mathbb{T}$

   $H : (\mathsf{VAR}\ (\mathsf{Ext}\Gamma' t1 \vdash n : t)$

   $H0 : \mathcal{TS}[\![\Gamma]\!]$

   Now, we proceed by cases on the natural number index n.

   NewDestruct n.

   (a) The first case, when n is 0. In this case, based on the hypothesis H, it easily follows that $t1 = t$, by examining the possible derivations. Thus, we assert a new goal $t1 = t$ and prove it by *Inversion*: *Assert* (*t*=*t1*). *Inversion H*.

   Now we can use this equality to rewrite all *t*s into *t1*s (using the tactic *Subst*) and simplify our assumptions and goals by issuing the following commands: *EAuto*.*Subst t*.

   $\Gamma' : \mathbb{G}$

   $t1 : \mathbb{T}$

   $H : (\mathsf{VAR}\ (\mathsf{Ext}\ \Gamma'\ t1 \vdash O : t1)$

   $H0 : \mathcal{TS}[\![\mathsf{Ext}\ \Gamma'\ t1]\!]$

   Recall, that the goal we are proving now is $\mathcal{T}[\![t1]\!]$. Now, the assumption *H0* has the type $\mathcal{TS}[\![\mathsf{Ext}\ \Gamma'\ t1]\!]$, which can be simplified by simply unfolding the definition of the meanings of type assignments to obtain the product:

H0 : $\mathcal{TS}[\![\Gamma']\!] \times \mathcal{T}[\![t1]\!]$.

Now, to obtain the goal, we only need to project the second part of *H0*:

EApply (*Snd H0*).

(b) There remains the one final case when the variable n is of the form $S\ m$. This case will be computed by making a recursive call to the function *lookUp*.

However, before such a recursive call can be made, we must have the appropriate judgment to give it to as an argument. Thus, we first assert a new goal that (*VAR* $\Gamma' \vdash m : t$), which is easily proved by inversion:

Assert (*VAR t0* $\vdash$ *m* : *t*). *Inversion H. Trivial.*

Then use this newly proved assumption, named *H1*, as one of the arguments to the recursive call of *lookUp*. One final step is to provide the runtime value of type $\mathcal{TS}[\![\Gamma']\!]$, which is obtained, as in the previous case by projecting, this time the first element, from the assumption (*H0* : $\mathcal{TS}[\![\Gamma']\!] \times \mathcal{T}[\![t1]\!]$).

Apply (*lookUp t0 m t H1* (*Fst H0*)).

The recursive call is accepted because the index argument *m* is structurally smaller than the initial argument *n*, which allows Coq to prove termination of *lookUp*:

Defined.

Having discharged all the cases, the interactive theorem prover states that all the goals have been proved. The command Defined instructs the prover to accept the proof term generated in the preceding interactive session as a definition for the function *lookUp*. Since it is defined using recursion, Coq makes sure before accepting the definition that *lookUp* always terminates.

The first thing one notices when examining the definition of *lookUp*, whether in terms of the interactive proof script or in terms of the generated code, is the large amount of "logical book-keeping." One example of this is the first case which we had to show that lookup in an empty environment leads to absurdity. Similarly, we had to assert and prove properties in other cases either to be able to project the 0-th variable or to make a recursive call to *lookUp:* both of these properties were easily obtainable from the definition of the typing judgment. These assertions end up being morally equivalent to various "generation lemmas" [5] one often proves when defining typing relations.

All this logical apparatus clutters up our definitions and makes the connection with the semantics stated in Section 2.2.1 rather obscure. So the question that presents itself immediately is *why not define the function lookUp by direct induction over the typing judgment (*VAR $\Gamma \vdash$ n *: t)?* Then, the assertion (*VAR* $\Gamma \vdash m : t$), for example, would be directly obtained from the inductive step, rather than having to be proved.

The reason why this does not work is a rather subtle but important feature of the Coq theorem prover.

In order to obtain a property of the system called *proof irrelevance* [], objects of kind *Prop* cannot be deconstructed (inductively, or by cases) in order to create/prove objects of kind *Set*. In other words, computational objects that live in the Set universe cannot depend on the structure of the proof objects in *Prop*, since the actual structure of a proof should be irrelevant: all proofs of the property P are equally valid.

The advantage of this principle is in the possibility of *extraction*, where the Coq system constructs programs (in Caml, Haskell, or Scheme) from its proofs or definitions. Under the extraction scheme, all objects of kind *Prop*, i.e., all proofs of properties are simply erased from the generated program. Thus, although the Coq term for *lookUp* is quite large, the extracted program is much more manageable, since most of the logical book-keeping disappears from the extracted program.

Still, it is possible to define functions such as *lookUp* more straightforwardly in Coq itself. The step required is to redefine the typing judgments to have kind *Set* rather than *Prop*.

Inductive $\mathbb{T}$: *Set* := *N* : $\mathbb{T}$ | *ArrT* : $\mathbb{T} \to \mathbb{T} \to \mathbb{T}$.
Inductive $\mathbb{E}$: *Set* := *Const* : *nat* $\to \mathbb{E}$ | *Var* : *nat* $\to \mathbb{E}$ | *Abs* : $\mathbb{T} \to \mathbb{E} \to \mathbb{E}$ | *App* : $\mathbb{E} \to \mathbb{E} \to \mathbb{E}$.
Inductive $\mathbb{G}$: *Set* := *Empty* : $\mathbb{G}$ | *Ext* : $\mathbb{G} \to \mathbb{T} \to \mathbb{G}$.

Notation "*t1* $\longrightarrow$ *t2*" := (*ArrT t1 t2*) (*at level* 65, *right associativity*).
Notation " λ *t* . *e* " := (*Abs t e*) (*at level* 40, *left associativity*).
Notation " *A* @ *B* " := (*App A B*) (*at level* 50, *left associativity*).
Notation " $\langle\rangle$ " := *Empty*.
Notation "*G* ;; *t*" := (*Ext G t*) (*at level* 60, *left associativity*).

Inductive *HasTypeVar* : $\mathbb{G} \to$ *nat* $\to \mathbb{T} \to$ *Set* :=
  HasTypeVar_Zero : (Γ:$\mathbb{G}$;*t*:$\mathbb{T}$)(*HasTypeVar* (Γ ;; *t*) *O t*)
| *HasTypeVar_Weak* : (Γ:$\mathbb{G}$;*n*:*nat*; *t*,*t'* : $\mathbb{T}$)(*HasTypeVar* Γ *n t*)$\to$ (*HasTypeVar* (Γ;;*t'*) (*S n*) *t*).
Notation " '*VAR*' $\Gamma \vdash$ *n* : *t* " := (*HasTypeVar* Γ *n t*).

Inductive *HasType* : $\mathbb{G} \to \mathbb{E} \to \mathbb{T} \to$ *Set* :=
  HasType_Const : (Γ:$\mathbb{G}$;*n*:*nat*)(*HasType* Γ (*Const n*) *N*)
| *HasType_Var* : (Γ:$\mathbb{G}$;*n*:*nat*;*t*:$\mathbb{T}$) (*VAR* $\Gamma \vdash$ *n* : *t*) $\to$ (*HasType* Γ (*Var n*) *t*)
| *HasType_Abs* : (Γ:$\mathbb{G}$;*t1*,*t2*:$\mathbb{T}$;*e*:$\mathbb{E}$) (*HasType* (Γ;;*t1*) *e t2*) $\to$ (*HasType* Γ (λ *t1* . *e*) (*t1* $\longrightarrow$ *t2*))
| *HasType_App* : (Γ:$\mathbb{G}$;*t1*,*t2*:$\mathbb{T}$;*e1*,*e2*:$\mathbb{E}$) (*HasType* Γ *e1* (*t1*$\longrightarrow$*t2*)) $\to$ (*HasType* Γ *e2 t1*) $\to$
    (*HasType* Γ (*e1* @ *e2*) *t2*).
Notation " '*EXP*' $\Gamma \vdash$ *e* : *t* " := (*HasType* Γ *e t*).

Figure A.2: New syntactic definitions for L_0, where judgments are in *Set*.

A.1.4 Set Judgments

Figure A.2 details the changes to the syntactic definitions that need to be made. The only change is in the type declaration of the type families *HasTypeVar* and *HasType*: they are declared as returning *Set* rather than *Prop*. This means, in particular, that they are no longer simply logical propositions, but also *"runtime values."* Now, let us examine a new, simpler definition of the function *lookUp*:

Definition *lookUp* : (Γ:$\mathbb{G}$)(*n*:*nat*)(*t*:$\mathbb{T}$)(*VAR* $\Gamma \vdash n : t$)→($\mathcal{TS}$ $[\![\Gamma]\!]$) → ($\mathcal{T}$ $[\![t]\!]$).

Now, rather than by general recursion on the index, we can immediately define the function by induction on the typing judgment argument (*VAR* $\Gamma \vdash n : t$). If we name the judgment argument *H*, then the following interactive commands to the theorem prover set up the definition by induction on *H*:

Intros. *Induction H*.

The definition is now split into two cases, yielding two goals:

1. The case for the base judgment, the runtime environment argument has the type $\mathcal{TS}$ $[\![\Gamma'; t]\!]$. We can immediately obtain the goal $\mathcal{T}$ $[\![t]\!]$ by projecting the second value from the runtime environment *H0*, which is a pair:

   EApply (*snd* ? ? *H0*).

2. In the case for the weakening judgment, we obtain the premises by the induction hypothesis:

   HrecH : ($\mathcal{TS}$ $[\![\Gamma']\!]$)→($\mathcal{T}$ *t*).

   H0 : $\mathcal{TS}$ $[\![\Gamma'$,*t*' $]\!]$

   What remains now is to apply the induction hypothesis to a smaller runtime environment:

   EApply (*HrecH* (*fst* ? ? *H0*)).

Defined.

To finish off, we can easily define the function *eval* which interprets typing judgments of expressions:

Definition *eval* : (Γ:$\mathbb{G}$; *e*:$\mathbb{E}$; *t*:$\mathbb{T}$)(*EXP* $\Gamma \vdash e : t$) → ($\mathcal{TS}$ $[\![\Gamma]\!]$) → ($\mathcal{T}$ $[\![t]\!]$).

This function is defined by recursion on the fourth argument, the typing judgment *H* of type (*EXP* $\Gamma \vdash e : t$), which is analyzed by cases:

Fix 4. *Intros*. *NewDestruct H*.

There are four cases:

1. In the case for integer constants, we have the integer constant *n* available as one of our assumptions. The goal to be proved is of type $\mathcal{T}[\![N]\!]$, which simplifies to *nat*. Thus to prove the goal we only need to exhibit a natural number, in particular the number *n*.

   EApply n.

2. In the case for variables, two assumptions are interesting, namely the variable sub-judgment *H* and the variable index *n*:

   H0 : (Var$\Gamma \vdash n$: *t*)

   n : *nat*

With these assumptions, we can make a call to the auxiliary function *lookUp*, previously defined:

Apply (*lookUp* Γ *n t H0*).

3. The function case creates as its result a function, the body of which is evaluated in an expanded runtime environment:

   Simpl. *Intros*. *EApply* (*eval* (Γ;;*t1*) *e t2 h* (*H0*,*H*)).

4. Finally, for the application case we compute the function value *fun*, as well as the argument value *arg* by recursive calls to *eval*. Then, *fun* is applied to *arg* to obtain the meaning of the application.

   LetTac fun := (*eval* Γ *e1* (*t1*⟶*t2*) *h H0*). *LetTac arg* := (*eval* Γ *e2 t1 h0 H0*). *EApply* (*fun arg*). Defined.

A.1.5 Program Extraction: Tagless Interpreters

We have mentioned Coq's ability to perform *program extraction* from its theorems and definitions. Extraction is fully automatic. The user only need specify some general parameters, such as for what target language (Haskell, OCaml, Scheme) extraction is performed, and simply indicate a Coq definition that should be extracted:

Recursive Extraction lookUp.

When issued this command, the theorem prover performs automatic extraction and prints out the text of the generated program, data-types, function definitions and all.

```
module Main where
import qualified Prelude
__ = Prelude.error "Logical or arity value used"
data Nat = O | S Nat
data Prod a b = Pair b a
fst p = case p of Pair x y → x
data Typ = N | ArrT Typ Typ
data TS = Empty | Ext TS Typ
lookUp gamma n t h0 =
  case gamma of
    Empty → Prelude.error "absurd case"
    Ext t0 t1 →
      (case n of O → (case h0 of Pair f e → e)
                 S n0 → lookUp t0 n0 t (fst h0))
```

Figure A.3: Extraction of *lookUp* (*Prop*-based judgments) as a Haskell function.

We may compare the two Haskell programs generated by extraction from the *Prop*- and *Set*-based implementations (Figure A.3 and Figure A.4, respectively).

Note that in Figure A.3 there is no trace of typing judgments. The function `lookUp` takes only three

```
module Main where
import qualified Prelude
data Nat = O | S Nat
data Prod a b = Pair b a
fst p = case p of Pair x y → x
snd p = case p of Pair x y → y
data Typ = N | ArrT Typ Typ
data TS = Empty | Ext TS Typ
data HasTypeVar =   HasTypeVar_Zero TS Typ
                  | HasTypeVar_Weak TS Nat Typ Typ HasTypeVar
lookUp gamma n t h h0 =
  let
    f t0 n0 t1 h1 h2 =
      case h1 of
        HasTypeVar_Zero gamma0 t2 → snd h2
        HasTypeVar_Weak gamma0 n1 t2 t' h3 → f gamma0 n1 t2 h3 (fst h2)
  in f gamma n t h h0
```

Figure A.4: Extraction of *lookUp* (*Set*-based judgments) as a Haskell function.

arguments: the type assignment, the index number and the runtime environment. Now, a combination of these arguments could be given to `lookUp` so that the resulting combination is not well-typed (i.e., there is no *VAR* judgment in the original Coq definition).

In these cases, the extracted program (e.g., line 11) uses the Haskell `error` value. These are cases that were defined by the *Absurd* tactic in the original – if the terms are well typed these cases should never occur.

By contrast, the *Set*-based typing judgments (Figure A.4) are extracted as a Haskell data-type `HasTypeVar`. Furthermore, the function `lookUp` takes a `HasTypeVar` as its fourth argument and pattern matches over it. This means that only well-typed judgments are analyzed and that there are no *absurd* cases. The price we pay for the *Set*-based definition is that the additional data-type `HasTypeVar` must be passed around in the extracted program and analyzed. This could result in potential runtime penalties.

An even more serious problem is present in the extracted problems in both styles of implementation: the programs extracted need not be, and usually *are* not, well typed in Haskell. The function `lookUp` (Figures A.3 and A.4) XXX is a case in point: each time around the recursive loop the runtime ' environment has a different type. This is because the type-system of Haskell is less expressive than the type system of Coq: Haskell rejects some well-typed Coq programs, even though they never cause runtime type errors.

For these extracted programs to be successfully compiled, the Haskell type checker must be turned off.[3] Since Haskell cannot type-check the program, the user must rely on the correctness of Coq's extraction

[3]Coq extraction for Objective Caml inserts the appropriate calls to the casting function, `Obj.cast` where it detects that Caml's type system is inadequate. This feature is not yet implemented for Haskell extraction.

algorithm to assure that the programs do not go wrong at runtime.[4]

A.2 Do We Need a Different Meta-Language?

With the experience described above one might ask: *Is Coq an adequate meta-language for our purposes of generating safe and efficient tagless interpreters? What additional/different features would be desirable in such a meta-language?*

1. Dependent types, especially inductive families, are quite useful in representing typing judgments, and providing clean, direct implementations of object-language semantics. Furthermore, the interactive theorem proving interface seems to be a very practical way of generating/writing programs.

2. However, a straightforward implementation in Coq along the lines shown for L_0 may be inadequate for many practical languages that do not enjoy the property of strong normalization. The Calculus of Inductive Construction, on which Coq is based, is a strongly normalizing calculus. That means, for example, that we have an object language with non-termination and/or arbitrary recursion, we cannot use Coq's function space to model the function space of object programs. While it is possible to develop domain theory in Coq, extraction of such definitions would not necessarily yield useful artifacts.

   An ideal meta-language would find a way to combine non-termination (and maybe other effects) with dependent types in some useful and manageable way.

3. Program generation via extraction may look good but introduces a number of problems:

   (a) If we use *Prop*-based implementation, considerable difficulties emerge with implementation of interpreters. First, we cannot define the meanings of programs (which live in *Set*) directly by induction/cases over typing judgments (which live in *Prop*). This can sometimes be circumvented by proving a number of "generation lemmas," but those lemmas become increasingly difficult to prove and use the more complex the language we are interpreting. Second, various "logical book-keeping" distracts from the clarity of definitions and obscures their connection to the semantics.

   (b) If we use *Set*-based judgments, the interpreters generated by extraction are neither really tagless, nor are they typable in Haskell. This leads to both a loss of performance, as well as to a

[4] In the writing of this chapter the author has discovered a rather unpleasant bug in Cog 7.4 extraction, so the issue, if anecdotal, is by no means irrelevant.

loss of reliability – we must rely on the correctness of the program extraction rather than the host language type system.

If we are to combine the simplicity and ease of (b) with the efficiency of (a), we would obtain quite a satisfactory implementation. Is this possible? Fortunately, it is, if we abandon program extraction in favor of *meta-programming by staging.* Then, we can use the (b) style to define interpreters, but use staging to perform all tagging-like operation (deconstruction of typing judgments) before the runtime of a particular object-language program (more about this later).

Bibliography

[1] Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, 1 May 1995.

[2] Lennart Augustsson. Cayenne – a language with dependent types, April 2004. Available from `http://www.cs.chalmers.se/ augustss/cayenne`.

[3] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming*, Gothenburg, 1999. Available online from www.cs.chalmers.se/~augustss/cayenne/interp.ps.

[4] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, SIGPLAN Notices 37(9). ACM Press, October 2002.

[5] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1991.

[6] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[7] Alan Bawden. Reification without evaluation. In *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM Press, 1988.

[8] Alan Bawden. Quasiquotation in LISP. In O. Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, San Antonio, January 1999. University of Aarhus, Dept. of Computer Science. Invited talk.

[9] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.

[10] Zine El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. A categorical analysis of multi-level languages (extended abstract). Technical Report CSE-98-018, Department of Computer Science, Oregon Graduate Institute, December 1998. Available from [98].

[11] Nikolaj Bjørner. Type checking meta programs. In *Workshop on Logical Frameworks and Meta-language*, Paris, September 1999.

[12] Corrado Böhm. The CUCH as a formal description language. In T. B. Steel, Jr, editor, *Formal Language Description Languages for Computer Programming, Proceedings of an IFIP Working Conference*, 1964.

[13] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagaciones Mathematische*, 34:381–392, 1972. This also appeared in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75, No. 5.

[14] N. G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. In *Proceedings of the Koninklijke Nederlandse Akaemie van Wetenschappen*, volume 81, pages 348–356, Amsterdam, September 1978. This paper was dedicated to A. Heyting at the occasion of his 80th birthday on May 9, 1978.

[15] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. *Lecture Notes in Computer Science*, 1853, 2000.

[16] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[17] L. Cardelli. Phase distinctions in type theory. Unpublished manuscript., 1998.

[18] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 275–286, Uppsala, Sweden, August 2003.

[19] James Cheney and Ralf Hinze. Phantom types. Available from `http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf`, August 2003.

[20] Robert L. Constable, S. Allen, H. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[21] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, 1986.

[22] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, Lecture Notes in Computer Science. Springer-Verlag, 1990.

[23] Thierry Coquand. A calculus of constructions. Privately circulated., November 1986.

[24] Thierry Coquand and Gérard Huet. A theory of constructions. Presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis, June 1984.

[25] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 233–248, N.Y., September 27–29 1999. ACM Press.

[26] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP) , Baltimore, Maryland, USA*, pages 301–312, 1998.

[27] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[28] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer-Verlag / ACM Press, 1999.

[29] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, July 1996. IEEE Computer Society Press.

[30] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, January 1996.

[31] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[32] D. Delahaye. A tactic language for the system Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of *Lecture Notes In Computer Science*, pages 85–95. Springer-Verlag, November 2000. ftp://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/LPAR2000-ltac.ps.gz.

[33] Peter Dybjer. Inductively defined sets in Martin-Löf's set theory. In A. Avron, R. Harper, F. Honsell, I. Mason, and G. Plotkin, editors, *Workshop on General Logic*, February 1987.

[34] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[35] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[36] U. W. Eisenecker. Generative programming (GP) with C++. *Lecture Notes in Computer Science*, 1204, 1997.

[37] Ulrich W. Eisenecker and Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[38] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers and Control*, 2(5):45–50, 1971.

[39] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, 1999. IEEE Computer Society Press.

[40] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[41] Murdoch J. Gabbay. *A Theory of Inductive Definitions with α-equivalence: Semantics, Implementation, Programming Language*. PhD thesis, Cambridge University, August 2000.

[42] Steve Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. *ACM SIGPLAN Notices*, 36(10):74–85, October 2001.

[43] Eduardo Gimenez. A tutorial on recursive types in coq. Technical report, INRIA, March 1998.

[44] Eduardo Giménez. A tutorial on recursive types in Coq. Technical Report TR-0221, INRIA Rocquencourt, May 1998.

[45] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.

[46] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[47] Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

[48] Carsten Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[49] Carsten Gomard and Neal Jones. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[50] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1979.

[51] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[52] Thomas Hallgren. *Alfa - User's Guide*, 2001. Available at: http://www.cs.chalmers.se/~hallgren/Alfa/userguide.html.

[53] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings Symposium on Logic in Computer Science*, pages 194–204, Washington, June 1987. IEEE Computer Society Press. The conference was held at Cornell University, Ithaca, New York.

[54] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Summary in LICS'87.

[55] Robert Harper and Greg Morrisett. Compiling polymorphism using intentional type analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 130–141, New York, NY, January 1995. ACM.

[56] William Harrison, Timothy Sheard, and James Hook. Fine control of demand in haskell. In *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer-Verlag, 2002.

[57] Susumu Hayashi. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 701–730. Springer-Verlag, September 1991.

[58] Susumu Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109(1/2):174–210, 15 February/March 1994.

[59] Susumu Hiyashi and Nakano Hiroshi. *PX, a Computational Logic*. MIT Press, 1988.

[60] W. A. Howard. The formulæ-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, London, 1980. Reprint of a manuscript written 1969.

[61] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, New York, 1980. A version of this paper was privately circulated in 1969.

[62] Liwen Huang and Walid Taha. A practical implementation of tag elimination. In preperation, 2002.

[63] John Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*. Springer-Verlag, February 1996.

[64] Report on the programming language haskell 98, February 1999.

[65] Mark P. Jones. *Qualified types: theory and practice*. PhD thesis, Keble College, Oxford University, 1992.

[66] Mark P. Jones. Type classes with functional dependencies. In *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000.*, volume 1782 of *Lecture Notes in Computer Science*, 2000.

[67] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993. Avaiable online from http://www.dina.dk/ sestoft.

[68] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *ACM Conference on LISP and Functional Programming*, pages 151–161, August 1986.

[69] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[70] Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In USENIX, editor, *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99), October 3–5, 1999, Austin, Texas, USA*, Berkeley, CA, USA, 1999. USENIX.

[71] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–69. ACM SIGACT and SIGPLAN, ACM Press, 1994.

[72] Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1997.

[73] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Walid Taha, editor, *Semantics, Applications and Implementation of Program Generation*, volume 1924 of *Lecture Notes In Computer Science*, pages 129–148, Montreal, Canada, 20 September 2000. Springer-Verlag.

[74] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North Holland, 1975.

[75] Conor McBride. Faking it: Simulating dependent types in haskell. *Journal of Functional Programming*, 12(4&5), July 2002.

[76] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.

[77] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.metaocaml.org/, August 2004.

[78] Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.

[79] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, 1990.

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

[81] Torben Mogensen. Inherited limits. In *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 189–202. Springer-Verlag, 1999.

[82] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[83] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[84] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[85] Aleksandar Nanevski and Frank Pfenning. Meta-programming with names and necessity. *Journal of Functional Programming*, 2003. Extended version of [86]. Submitted for publication.

[86] Aleksander Nanevski. Meta-programming with names and necessity. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

[87] Peter Naur (editor), et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 5(1):1–17, January 1963.

[88] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[89] Neal Nelson. *Type Inference and Reconstruction for First Order Dependent Types*. PhD thesis, Oregon Graduate Institute, 1994.

[90] Flemming Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, July 1985.

[91] Flemming Nielson. Correctness of code generation from a two-level meta-language. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming (ESOP 86)*, volume 213 of *Lecture Notes in Computer Science*, pages 30–40, Saarbrücken, March 1986. Springer.

[92] Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, December 1989.

[93] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[94] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[95] Flemming Nielson and Hanne Riis Nielson. A prescriptive framework for designing multi-level lambda-calculi. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 193–202, Amsterdam, June 1997. ACM Press.

[96] Bengt Nordström. Programming in constructive set theory: Some examples. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture, Portsmouth, NH*, pages 141–154, New York, 1981. ACM.

[97] Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Lof's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, New York, NY, 1990. Currently available online from first authors homepage.

[98] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html.

[99] Emir Pasalic. Meta-programming with typed object-language representations. In *GPCE'04*, 2004.

[100] Emir Pasalic, Tim Sheard, and Walid Taha. Dali: An untyped, cbv functional language supporting first-order datatypes with binders. Proofs and technical development in [101]., March 2000.

[101] Emir Pasalic, Tim Sheard, and Walid Taha. Dali: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report CSE-00-007, OGI, March 2000. Available from [98].

[102] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *The International Conference on Functional Programming (ICFP '02)*, SIGPLAN Notices 37(9), Pittsburgh, USA, October 2002. ACM.

[103] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages (formal development). Technical Report 02-006, OGI, 2002. Available from [98].

[104] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proc. of 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, Berlin, 1993.

[105] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5–6):607–640, 1993.

[106] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[107] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[108] Frank Pfenning and Rowan Davies. A judgmental reconstuction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[109] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.

[110] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, Mass., 1991.

[111] Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[112] Jonathan Rees, William Clinger, H. Abelson, N. I. Adams IV, D. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised^4 report on the algorithmic language Scheme. Technical Report AI Memo 848b, MIT Press, November 1992.

[113] Kristoffer H. Rose. Explicit substitution – tutorial & survey. Technical Report LS-96-3, BRICS, Universitty of Århus, october 1996. BRICS Lecture Series.

[114] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 116–129, La Jolla, California, 18–21 June 1995.

[115] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. Technical Report YALEU/DCS/TR-1211, Yale University, March 2001. revised September 2001.

[116] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 31(1):217–232, January 2002.

[117] T. Sheard, Z. Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, October 1999. USEUNIX.

[118] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Semantics, Applications, and Implementation of Program Generation : Second International Workshop*, volume 2196 of *Lecture Notes in Computer Science*, 2001.

[119] Tim Sheard. Universal meta-language. Available from the author., 2003.

[120] Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, 1999. USEUNIX.

[121] Tim Sheard, Zino Benaissa, Walid Taha, and Emir Pasalic. MetaML v1.1. http://www.cse.ogi.edu/PacSoft/projects/metaml, March 2001.

[122] Tim Sheard and Neal Nelson. Type safe abstractions using program generators. Technical Report CSE-95-013, Oregon Graduate Institute, 1995. Available from [98].

[123] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, July 2004.

[124] Tim Sheard, Emir Pasalic, and R. Nathan Linger. The ωmega implementation. Available on request from the author., 2003.

[125] Tim Sheard and Peter Thieman. Metaml on the run: A constrained type system for staged execution of open code. Available from authors., August 2004.

[126] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of LISP. In Richard L. Wexelblat, editor, *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, pages 231–270, New York, April 1993. ACM Press.

[127] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[128] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.rice.edu).

[129] Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM'00)*, January 2000.

[130] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, July 1998.

[131] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. Technical Report CSE-98-002, Oregon Graduate Institute, 1998. Available from [98].

[132] Walid Taha and Henning Makholm. Tag elimination – or – type specialisation is a type-indexed effect. In Subtyping and Dependent Types in Programming, APPSEM Workshop. INRIA technical report, 2000.

[133] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 257–275, 2001.

[134] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, volume 38, 1 of *ACM SIGPLAN Notices*, pages 26–37, New York, January 15–17 2003. ACM Press.

[135] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997. An extended and revised version appears in [137].

[136] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Department of Computer Science, Oregon Graduate Institute, January 1999. Extended version of [135]. Available from [98].

[137] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000. In Press. Revised version of [136].

[138] Robert D. Tennent. *Semantics of Programming Languages.* Prentice Hall, New York, 1991.

[139] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*. INRIA, 2003. http://pauillac.inria.fr/coq/doc/main.html.

[140] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 82–93, N.Y., September 18–21 2000. ACM Press.

[141] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[142] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[143] Stephanie Weirich. Type-safe cast: functional pearl. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 58–67, N.Y., September 18–21 2000. ACM Press.

[144] Benjemin Werner. (im)personal communication. The Coq mailing list., May 2000.

[145] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[146] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, volume 38, 1 of *ACM SIGPLAN Notices*, pages 224–235, New York, January 15–17 2003. ACM Press.

[147] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 249–257, Montreal, Canada, 17–19 June 1998.

[148] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, January 1999. ACM.

Biographical Note

Emir Pašalić was born on June 20, 1977, in Zagreb, Socialist Federal Republic of Yugoslavia (today Croatia). He arrived in the United States in 1993, seeking to continue his education. He studied computer science and completed a B. S. degree at The Evergreen State College in Olympia, Washington.

After completing his undergraduate studies, he started his Doctoral studies at the (then) Oregon Graduate Institute of Science and Technology in Beaverton, Oregon. He has worked with his advisor, Prof. Tim Sheard, on a number of issues relating to meta-programming. His interest have included languages with support for higher-order abstract syntax, semantics of staged computation, and type theoretical approaches to building efficient interpreters.

During his stay at OGI, Emir interned for several months with the Galois Connections, Inc., an OGI-Pacsoft spin-off firm specializing in software engineering using functional programming languages. He worked on designing and implementing a large scale legacy code translator for hardware testing programs.