Henk: a typed intermediate language

Simon Peyton Jones
University of Glasgow and Oregon Graduate Institute

Erik Meijer
University of Utrecht and Oregon Graduate Institute

May 20, 1997

Abstract

There is growing interest in the use of richly-typed intermediate languages in sophisticated compilers for higher-order, typed source languages. These intermediate languages are typically stratified, involving terms, types, and kinds. As the sophistication of the type system increases, these three levels begin to look more and more similar, so an attractive approach is to use a single syntax, and a single data type in the compiler, to represent all three.

The theory of so-called *pure type systems* makes precisely such an identification. This paper describes Henk, a new typed intermediate language based closely on a particular pure type system, the *lambda cube*. On the way we give a tutorial introduction to the lambda cube.

1 Overview

Many compilers can be divided into three main stages. The *front end* translates the *source language* into an *intermediate language*; the *middle end* transforms the intermediate-language into a more efficient form; and the *back end* translates the intermediate language into the *target language*.

In the past, intermediate languages for source languages with rich type systems have usually been un-typed. The compiler first type-checks the source program, and then translates the program to the intermediate language, discarding all the type information. After all, the type checking simply ensured that the program would not "go wrong" at run-time, and once that is checked there is no further use for types.

Recently, however, there has been increasing interest in *typed* intermediate languages (Peyton Jones [1996]; Peyton Jones et al. [1993]; Shao & Appel [1995]; Tarditi et al. [1996]). There are several motivations for such an approach: the compiler may be able to take advantage of type information to generate better code; it may be desirable to treat types as values at run-time, in which case it is necessary to maintain accurate compile-time types; and the compiler can check its own activity (if desired) by checking the type-correctness of the intermediate program. We elaborate each of these points in Section 2.1.

This paper will be presented at the June 1997 Workshop on Types in Compilation.

This paper describes the design of a new, typed intermediate language, Henk, designed for compilers for purely-functional languages. It has the following distinctive features:

- Henk *is based directly on the lambda-cube*, an expressive family of typed lambda calculi. We have found a shortage of introductory material about the lambda cube, so we present a tutorial in Section 3.
- Henk *is a small language* — there are only seven constructors in the data type of expressions. Even so, it is a *real* language, in the sense that it is rich enough to use as a compiler intermediate language.
- Because of its lambda-cube heritage, Henk *uses a single syntax for terms, types, and kinds.* Better still, compilers for Henk can use a single data type to represent all three levels. This leads to considerable economy in both the syntax of the language, and the utility functions of the compiler itself.
- Henk *has an explicit concrete syntax.* Intermediate languages are typically expressed only as a data type in a particular compiler. Giving a concrete syntax is an apparently trivial step, but we believe it is an important one, and it is one to which compiler-writers often pay little attention. A compiler front end can produce Henk to be consumed by the back end of a different compiler; or to be transformed by some external program before being fed back into the original compiler.

This paper introduces no new technical results. Rather, our main contribution is to build a bridge between recently-developed type theory and the compiler research community:

- We are the first to suggest using the lambda cube as the basis for a typed intermediate language. Why bother? We see two persuasive reasons:
  - It dramatically reduces the number of data types, and the volume of code, required in the compiler. For example, the Glasgow Haskell Compiler (GHC) has separate data types for terms, types,

and kinds, and separate algorithms for parsing, printing, typing, and transforming them. Collapsing the three levels gets rid of all this duplication.

  - It is easier to accommodate new developments in the type system of the source language, because the lambda cube's type language is already as expressive as its term language.

- We give a tutorial on the lambda cube, emphasising aspects relevant to compiler builders. (Most of the literature is relatively recent — post 1988 — and written from the point of view of theorists.)

Our initial focus is on *non-strict* languages, but we hope to make Henk neutral with respect to the strict/non-strict question. That is, rather than having two variants of Henk (one strict, one non-strict), we hope to have a single language that treats both styles as first class citizens, and allows free mixing of the two in a single program.

We assume familiarity with the lambda calculus in general, and with the second-order lambda calculus, also called $F2$, in particular.

2 Background and motivation

2.1 Type-directed compilation

It has been recognized for some time that maintaining type information right through to code generation, and beyond, can be beneficial. Specifically:

- Accurate type information can guide compiler analyses, and transformations.

  For example:

  - Strictness analysis over non-flat domains has to be guided by type information (Peyton Jones & Partain [1993]).
  - A compiler may be able to use more efficient representations of data values if it knows their types. For example, an integer can be represented by the integer itself rather than a pointer to a box containing the integer (either in a strict language, or in a lazy one in contexts where the integer is certainly evaluated). The price to be paid is that such unboxed integers cannot be passed to polymorphic functions, since their representation differs from the simple pointer that polymorphic code typically expects. Guided by type information one can specialise the polymorphic function to the unboxed types at which it is used (Peyton Jones & Launchbury [1991]; Tarditi et al. [1996]).
  - Transformations that remove intermediate data structures, often called deforestation or fusion transformations, rely heavily on types to guide the transformation. In some cases the very correctness of the transformation relies on parametricity, a property of well-typed polymorphic functions (Gill, Launchbury & Peyton Jones [1993]; Hu, Iwasaki & Takeichi [1996]; Launchbury & Sheard [1995]).

- Traditional static type checking is performed completely at compile time. In more sophisticated settings, however, it may be useful to postpone some type checking until run time. For example:

  - Some reasonable programs cannot readily be expressed in a static type systems, notably ones involving meta-programming. There are many proposals for incorporating a type `Dynamic` in an otherwise statically-typed language, but all involve some run-time check that the type of a value matches an expected type.
  - Rather than statically specialise a polymorphic function for the types at which it is called, one can pass the type as an explicit argument, so that the function can behave appropriately (Harper & Morrisett [1995]).
  - "Tag-free" garbage collectors need some run-time type information to guide them (Tarditi [1996]; Tolmach [1994]).

  All these applications require accurate type information to be available at run-time, and hence at compile time too.

- Debugging a compiler can be a nightmare. The only evidence of an incorrect transformation can be a segmentation fault in a large program compiled by the compiler. Identifying the cause of the crash, and tracing it back to the faulty transformation, is a long, slow business. If, instead, the compiler type-checks the intermediate-language program after every transformation, the huge majority of transformation bugs can be nailed in short order. It is quite difficult to write an incorrect transformation that is type correct! Furthermore, the program cannot crash if it is type correct, so even incorrect transformations can only give an unexpected result, not a crash, which is usually much easier to trace.

  We call the Henk type-checker "Core Lint". If the compiler is correct, Core Lint does nothing useful, and is switched off by default. In our experience, its ability to localise compiler bugs would by itself justify the use of a typed intermediate language.

Compilers that maintain and use type information throughout the compiler have come to be called "type-directed". Standard ML of New Jersey has been type-directed for some time (Shao & Appel [1995]), but its internal type system is monomorphic, so it cannot track types through polymorphic functions. Since 1990 the Glasgow Haskell Compiler (GHC) has used a language based on the second-order lambda calculus as its intermediate language (Peyton Jones [1996]; Peyton Jones et al. [1993]).

More recently, the TIL compiler uses a considerably more sophisticated (and complicated) intermediate language capable of expressing intensional polymorphism, including recursive functions at the level of types (Morrisset [1995]; Tarditi et al. [1996]). TIL uses a stratified type system, which simplifies the proof theory, but it does lead to considerable duplication in compiler transformations (Tarditi [1996]). We speculate that the lambda cube might provide a theoretically sound way to get the best of both worlds.

Shao is also developing a typed intermediate language, FLINT, with similar goals to TIL's (Shao [1996]).

2.2 Implicit vs explicit typing

It is important to distinguish the typing requirements of a source language and an intermediate language.

Almost all source languages are, to some degree, *implicitly* typed. There is a continuum between full type inference (where no type information is given by the programmer) and type checking (where all type information is given). The type system of a source language is usually a delicate compromise of expressiveness and type inference. The more expressive the type system, the more guidance (in the form of type signatures and the like) must be given to the type checker.

On the other hand, it is highly implausible to have an implicitly-typed intermediate language, because it hard to ensure that every transformation preserves the delicate property that types can be inferred from the program source. Indeed, many transfomations do not (e.g. desugaring a `let` expression to a lambda abstraction applied to the right-hand side of the `let` definition). An intermediate language can, however, be *explicitly* typed. When the front end translates a source program into the intermediate language, it can decorate it with type information based on the results of type inference on the source program. If the compiler, for some reason, wants to check the type-correctness of an intermediate-language program, it should be a matter of type *checking* and not type *inference*. In practical terms, type checking does not involve unification or other sophisticated algorithms.

Furthermore, because it is explicitly typed, the intermediate language can have a much more expressive type system than the source language. For example, the intermediate language might permit arbitrary universal quantification in types, whereas a source language like ML or Haskell restrict universal quantification to the top level of a type[1].

2.3 Towards the lambda cube

While it is obvious enough in retrospect, it was a breakthrough when we realised in 1990 that the second-order lambda calculus was precisely what we needed to express and maintain type information in the intermediate language of a compiler. (Peyton Jones [1996] gives some examples.) A large body of theory and its design choices, could be pressed into immediate service.

With Henk, we aim to take the same idea one step further, by appropriating another body of theory, the lambda-cube, and adopting its design choices to structure the language.

[1] Haskell's type classes actually give rise to intermediate-language constants of rank-2 polymorphic type, a nice example of the way in which a language feature can make use of a type discipline which would be unworkable in its full generality.

Specifically, Henk goes beyond the second order lambda calculus in the following ways:

- It is elegantly parameterised. Simply by selecting or discarding type rules one can force Henk to be equivalent to the simply-typed lambda calculus, the second-order lambda calculus (Girard [1972]), its extension to higher-order kinds ($F\omega$) (Girard [1972]), or the calculus of constructions (λC) (Coquand & Huet [1988]). In Section 4 we show how we can also extract a predicative version of $F\omega$.
- As it happens, Haskell's type system allows type variables to range over type *constructors* (not merely types), so the generality of $F\omega$ is already required. Henk puts this extension on a firm theoretical foundation, whereas it is a somewhat *ad hoc* extension of GHC's earlier Core language.
- Henk provides a full lambda calculus at the level of types. Provided recursion is disallowed — a restriction easily expressed in the type system — evaluation of types is strongly normalising (i.e. guaranteed to terminate), something that is really the defining property of a type.

  This lambda calculus subsumes Haskell's type synonyms, which receive a rather *ad hoc* treatment in GHC, and it also permits us to explore more ambitious paths such as those suggested by TIL.
- Despite this extra richness, Henk is a very small language. The data type of expressions, for example, has only seven constructors. Better still, the very same syntax is used for expressions, types, and kinds. This economy is reflected:
  - in the type system by a single set of rules that say when a term is well-typed, when a type is well-kinded, and when a kind is well formed.
  - in the compiler by a single data type that represents terms, types and kinds; and a single set of utility functions to parse, print, type-check, and so on.

One might object that using the same compiler data type to represent terms and types would allow a buggy compiler to construct ill-formed terms, such as attributing the type `3` to a variable. With GHC's current structure, such a thing would be identified as ill-typed when compiling the compiler, because something from the datatype representing terms cannot be used in place of something from the datatype representing types.

GHC's current structure may prevent the compiler from accidentally constructing *some* bogus terms, but it does not eliminate *all* of them. For example, `3 True` is a legal value in the datatype of terms, but of course it is ill-formed. Instead we rely on Core Lint (Section 2.1) to identify such bugs. Folding together the three levels does, however, postpone the detection of certain (actually rather unusual) errors from compiler-compilation-time to compiler-run-time.

$$
\begin{array}{rcll}
E & ::= & K & \text{Constant} \\
 & | & x & \text{Variable} \\
 & | & EE & \text{Application} \\
 & | & \lambda x : E.E & \text{Abstraction} \\
 & | & \Pi x : E.E & \text{Quantification}
\end{array}
$$

Figure 1: Syntax of Pure Type System expressions

3 Pure Type Systems and the Lambda Cube

So what should the new typed intermediate language look like? Fortunately, Barendregt has done all the hard work for us (Barendregt [1992]). Pure Type Systems (PTS) are an elegant way of presenting explicitly-typed lambda-calculi in a uniform way, and give us almost exactly what we want. However, the literature on pure type systems is mostly written from the perspective of theorists, and while much of it is excellent, it is not for the faint hearted. In this section we therefore begin with a tutorial on Pure Type Systems, using a very small expression language. In Section 5 we then elaborate this calculus into a real language.

3.1 The familiar core

The syntax of PTS expressions is given in Figure 1. The first four productions should be familiar: constans, variables, applications, and abstractions. The language is explicitly typed, so that the variable bound by a lambda abstraction is annotated with its type. The fifth production, Π-abstraction, is a key idea of PTS, and as we will see shortly subsumes both function arrow and universal quantification.

3.2 Mixing terms and types

The unusual feature about the PTS world is that the type that decorates the bound variable of a lambda abstraction is simply another expression. That is, *types have the very same syntax as terms.*

This seems like an attractive idea. After all, like terms, the conventional syntax of types includes constants (e.g. `Int`), type variables, applications (e.g. `Tree Int`), abstractions (in the form of type synonyms), and (in a polymorphic system) a binding construct, namely universal quantification. Furthermore, the second order lambda calculus requires abstractions and applications of types to appear in terms.

In a PTS, a single form of abstraction and application suffices, at least from a syntactic point of view. For example, here is an expression written in F2:

$$\Lambda\alpha.\lambda\mathtt{x} : \alpha.\mathtt{id}\ [\alpha]\ \mathtt{x}$$

Λ introduces a type abstraction that binds the type variable α, which in turn is used as the type of `x`. In the body of the abstraction, the polymorphic identity function, of type $\forall\alpha.\alpha \to \alpha$, is applied to α and `x`. We use square brackets to indicate type application.

In our PTS language, the same abstraction and application forms serve for both types and terms, so we can rewrite the expression thus:

$$\lambda\alpha : \star.\lambda\mathtt{x} : \alpha.\mathtt{id}\ \alpha\ \mathtt{x}$$

Notice that this decision forces us to attribute a type to the variable α. The type of a type is called its *kind*, and $\star$ is a kind constant, usually pronounced "type". Thus "$\alpha : \star$" simply says that α is of kind "type"; that is, α is a type variable. The question that begs to be asked is this: is $\star$ the only kind? The answer is "yes" for $F2$, and "no" for $F\omega$. In $F\omega$, type variables can range over type constructors as well as over types; indeed, this is precisely what distinguishes it from $F2$. For example, in $F\omega$ we might write:

$$(\lambda\mathtt{m} : \star \to \star.\lambda\mathtt{x} : \mathtt{m\ Int}\ldots)\ \mathtt{Tree}$$

Here, the first lambda abstracts over `m`, whose kind is $\star \to \star$. The second lambda abstracts over values of type `m Int`. The whole abstraction is applied to a type constructor `Tree`, whose kind is presumably $\star \to \star$.

In short, even if the syntax had not forced us to attribute a type to α, the move from $F2$ to $F\omega$ would have done so. Is the extra power of $F\omega$ required in a compiler intermediate language? Clearly this depends on the source language, but the extra power is certainly required for Haskell, whose type system explicitly includes higher-kinded type variables[2], most particularly to support *constructor classes* (Jones [1995]).

So far this all seems attractive, but there are at least three worries:

1. How should we interpret function arrow, "$\to$", in the language of types? As a constant? Perhaps, but it is a very special one indeed, because it has an intimate relationship with abstraction and application at the term level.
2. What is to play the role of universal quantification, "$\forall$", in the language of types? (It does not take much experimentation to convince oneself that λ is inappropriate for this purpose.)
3. Now that the types and values are mixed up together, how can we be sure that the resulting expression still makes sense? That is, can we give sensible type rules for the language?

These questions are all elegantly resolved by the fifth form of expressions given in Figure 1, to which we turn our attention next.

3.3 Notation

Before we do, it will be helpful to establish some terminology. We have identified three levels so far: *terms*, *types*, and *kinds*. An *expression*, described by the syntax of Figure 1, can denote a term, a type, or a kind. We call these three levels *sorts*, so that we might say that an expression is of

[2]This was an innovation in Haskell 1.3; earlier versions of Haskell did not have higher-kinded type variables.

sort `Term`, or of sort `Type`, or of sort `Kind`. More commonly, though, we simply say that an expression is a term, or is a type, or is a kind.

Unfortunately it is very hard to avoid using the word "type" for multiple purposes. In particular, note the difference between sort `Type` and kind type. For example, both $\texttt{Int} : \star$ and $\texttt{Tree} : \star \rightarrow \star$ are of sort `Type`; but only `Int` is of kind type ($\texttt{Int} : \star$).

Each well-formed term has (belongs to) a type, and each well-formed type has a kind. A *type system* specifies precisely which expressions are well-formed and which are not. In general, a PTS may have more than three levels (or even an infinite number), but in this paper we study a particular family of PTSs called *the lambda cube*. The type system of the lambda cube ensures that no more than three levels are required, apart from a solitary constant, $\Box$, at the fourth level, as we shall see.

When writing example programs, we will use typewriter font for term variables (e.g. `x`) and all constants (e.g. `+`, `Int`), and Greek font for type variables (e.g. α, β). When writing program schemes (for example in type rules) we will use x, y, z to range over variables (of all sorts), and A, B, C, a, b, c to range over expressions (of all sorts). Generally, A will be of a sort one higher than a; thus we might say that "the term a has type A". Finally, we use s, t to range over the constants $\star$ and $\Box$.

3.4 Quantification

The fifth production in the syntax of PTS expressions (Figure 1) introduces the *dependent product*, Π. There are many essentially-equivalent ways of interpreting the expression $\Pi x : A.B$, but for our present purposes the most useful one is this:

> $\Pi x : A.B$ is the type of functions from values of type A to values of type B, in which the result type B may perhaps depend on the value of the argument, x.

From this definition it is immediately clear that Π subsumes the function arrow $\rightarrow$:

$A \rightarrow B$	is an abbreviation for	$\Pi\_ : A.B$

where we use the underscore symbol "_" to denote an anonymous variable . (We could instead say "$\Pi x : A.B$ where x does not occur free in B", but "$\Pi\_ : A.B$" is briefer.)

What is not so obvious is that Π also subsumes universal quantification, $\forall$. Consider the type $\Pi\alpha : \star.A$, where A is a type. This type is the type of functions from values of kind $\star$ (that is, types) to values of type A (that is, terms), where the type A may mention α. *But that is precisely the interpretation we would give to* $\forall\alpha.A$*!* In short,

$\forall\alpha.A$	is an abbreviation for	$\Pi\alpha : \star.A$

For example, consider the K combinator, defined thus: $K\ x\ y\ =\ y$. In F2 we would write the typing judgement for K's body like this:

$$\vdash (\Lambda\alpha\Lambda\beta.\lambda\texttt{x} : \alpha.\lambda\texttt{y} : \beta.\texttt{y}) : (\forall\alpha\beta.\alpha \rightarrow \beta \rightarrow \beta)$$

In a PTS we would write the judgement like this[3]:

$$\vdash (\lambda\alpha : \star.\lambda\beta : \star.\lambda\texttt{x} : \alpha.\lambda\texttt{y} : \beta.\texttt{y}) : (\Pi\alpha : \star.\Pi\beta : \star.\alpha \rightarrow \beta \rightarrow \beta)$$

Voilà! With one blow, Π deals with two of the three worries at the end of the Section 3.2. However, it does so at the price of making the third worry even more worrying. For example, what is to stop us from writing types like this one?

$$\Pi\texttt{x} : \texttt{Int.if x>3 then Int else Bool}$$

This is the type of functions from values of type `Int` to a result of type `Int` if the argument is greater than `3`, or `Bool` otherwise. Type checking may now require arbitrary computations at the term level!

The PTS framework allows us to answer the question in one of two ways:

1. either we can arrange for strange types like the one above to be ill-formed;
2. or we can decide that we like the expressiveness that it gives, and permit it.

The latter choice is equivalent to adopting the calculus of constructions (Coquand & Huet [1988]).

The PTS framework allows terms and types to "mix". We ensure that we can only construct expressions that "make sense" by means of a type system, which is what we discuss next.

3.5 The lambda cube type system

We write typing judgements in the conventional way. The judgement:

$$\Gamma \vdash E : A$$

is read "in environment Γ the expression E has type A". The environment gives types for the free variables of the expression E. So, for example, we could correctly state:

$$\{\texttt{Int} : \star, \texttt{+} : \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}\}$$
$$\vdash$$
$$(\lambda\texttt{x} : \texttt{Int.+ x x}) : \texttt{Int} \rightarrow \texttt{Int}$$

The type rules for the lambda cube are given in Figure 2 and 3. The VAR rule should be quite familiar; it simply says that if the environment Γ attributes the type A to x, then we can conclude that $x : A$. The premise checks that the type A of x is itself well formed. The context Γ is a sequence, not a set, with inner bindings to the right of outer ones. The weakening rule, WEAK, allows us to throw away irrelevant bindings (but checking that they are each well formed); it is usually applied as often as necessary just before the VAR rule.

[3]Remember, "$A \rightarrow B$" is just an abbreviation for "$\Pi\_ : A.B$"

$$\frac{}{\vdash \star : \Box} \qquad \text{(STAR)}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B} \qquad \text{(WEAK)}$$

$$\frac{\Gamma \vdash f : (\Pi x : A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x := a]} \qquad \text{(APP)}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \qquad \text{(LAM)}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t \qquad \vdash s \rightsquigarrow t}{\Gamma \vdash (\Pi x : A.B) : t} \qquad \text{(PI)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash a : B} \qquad \text{(CONV)}$$

Figure 2: Type rules for the Lambda Cube

The second rule, STAR, is also easy. It states that the constant $\star$ has super-kind $\Box$. This is where the hierarchy stops in the lambda cube. There is no typing rule for $\Box$ and hence it cannot appear explicitly in a program.

Things become more interesting when we meet the rule for applications, APP. In ordinary lambda calculus one usually sees a rule like this:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B} \qquad (APP_{F2})$$

Remembering that $A \to B$ is an abbreviation for $\Pi\_ : A.B$, it is easy to see the "ordinary" rule can be obtained by specialising rule APP with $x = \_$. The substitution of a for x in B does nothing, because in this special case x cannot occur in B — that is what the "_" meant.

The exciting thing is that the same rule, APP, also deals correctly with type applications. In $F2$, we have this rule for type applications (often called SPEC, since it specialises a polymorphic type):

$$\frac{\Gamma \vdash f : \forall \alpha.B}{\Gamma \vdash f\ [A] : B[\alpha := A]} \qquad (SPEC_{F2})$$

A few moments thought, remembering that $\forall \alpha.B$ is an abbreviation for $\Pi \alpha : \star.B$, should convince you that APP indeed subsumes $SPEC_{F2}$. This time the substitution of a for x in B in APP is vital, just as we must substitute A for α in B in $SPEC_{F2}$.

$$\frac{}{\vdash \star \rightsquigarrow \star} \qquad (\star \rightsquigarrow \star)$$

$$\frac{}{\vdash \Box \rightsquigarrow \star} \qquad (\Box \rightsquigarrow \star)$$

$$\frac{}{\vdash \Box \rightsquigarrow \Box} \qquad (\Box \rightsquigarrow \Box)$$

$$\frac{}{\vdash \star \rightsquigarrow \Box} \qquad (\star \rightsquigarrow \Box)$$

Figure 3: The $\rightsquigarrow$ judgement

Rule APP expects f to have a Π type. The next rule in Figure 2, LAM, shows how Π types are introduced in the first place. As before, it is helpful to compare it with the rules for $F2$. The $F2$ rules for value and type abstractions are:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash (\lambda x : A.b) : A \to B} \qquad (VLAM_{F2})$$

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash (\Lambda \alpha.b) : \forall \alpha.B} \qquad (TLAM_{F2})$$

The first of these states that if, assuming x has type A we can prove that the body, b, of the abstraction has type B, then the abstraction $\lambda x : A.b$ has type $A \to B$. Compare this with rule LAM in Figure 2; the first premise and the conclusion match the $F2$ rules directly. The second premise is more interesting. Its mission is to check that it is legitimate to abstract a variable $x : A$ from an expression of type B. It does this simply by requiring that the type in the conclusion is a legitimate type — that is, that it itself has a type. For example, the type abstraction $\Lambda \alpha.E$ is permitted iff its type, $\forall \alpha.\sigma$ is permitted, where $E : \sigma$.

Of course, this just begs the question. When, precisely, is a Π type a valid type? To answer that look at the PI rule. It specifies how to find the type of the expression $(\Pi x : A.B)$. The first premise checks that A is well formed, and finds its type, s. The second does the same for B, remembering to bring x into scope first. The third premise uses a new judgement form, $\rightsquigarrow$, which is also defined by the four axioms in Figure 3 . Since these axioms only involve constants, the $\rightsquigarrow$ judgement does no more than generate four copies of rule PI, each with its own values for s and t. In fact, the lambda cube describes a family of eight type systems, each defined by selecting a subset of the axioms for $\rightsquigarrow$, as we now discuss.

Let us first specialise rule PI with the axiom $\star \rightsquigarrow \star$. Then $s = \star$ and $t = \star$. What does this mean? It means that A and B must both be of kind $\star$; that is, they must both be types, and hence (moving back to rule LAM) we can abstract a term variable over a term expression. Indeed, we can read "$\star \rightsquigarrow \star$" as "terms can depend on terms". Furthermore, if $\star \rightsquigarrow \star$ is the only axiom for $\rightsquigarrow$, then these value abstractions are the *only* sort of abstractions we are allowed, so we have the simply-typed lambda calculus.

What is needed to allow type abstractions, which we need for the second-order lambda calculus, $F2$? Looking at rule LAM, we will need x to be a type variable, so A must be a

$$\frac{}{(\lambda x : A.B)\ C =_\beta B[x := C]} \quad (\mathrm{LAM}_{conv})$$

...plus the usual rules for symmetry, reflexivity, and transitivity.

Figure 4: Conversion rules for expressions

kind, so s must be $\Box$. However the body of a type abstraction is a term, so B is a type, and hence t is $\star$. Hence, to get $F2$ we need the axiom $\Box \leadsto \star$, which we can read as "terms can depend on types"[4].

To get $F\omega$ we need not only that A is a kind, but also that it can be a higher kind, such as $\star \to \star$. If $A = \star \to \star$, the first premise of PI requires that $\Gamma \vdash \star \to \star : s$, which in turn requires us to give the typing judgement for $\to$, that is, for Π. So we have to re-use rule PI. It is an almost immediate consequence that we require the axiom $\Box \leadsto \Box$ in order to conclude $\vdash \star \to \star : \Box$. Hence, $F\omega$ requires the axiom $\Box \leadsto \Box$, which we can read as "types can depend on types".

What would happen if we added the final axiom, $\star \leadsto \Box$? That would allow "types can depend on terms", taking us into the Calculus of Constructions (Coquand & Huet [1988]). This is swampy territory for compilers, so we stay well clear and avoid $\star \leadsto \Box$.

To summarise, we have the following equivalences:

System	Corresponding subset of $\leadsto$
Simply typed λ-calculus	$\{\star \leadsto \star\}$
$F2$	$\{\star \leadsto \star, \Box \leadsto \star\}$
$F\omega$	$\{\star \leadsto \star, \Box \leadsto \star, \Box \leadsto \Box\}$
Calculus of constructions	$\{\star \leadsto \star, \Box \leadsto \star, \Box \leadsto \Box, \star \leadsto \Box\}$

There are eight systems given by selecting $\star \leadsto \star$ and any subset of the remaining three axioms, which is what gives rise to the term "lambda cube". All eight systems make sense, but the ones just identified are the interesting ones.

3.6 Completing the type system

The final rule in Figure 2 is CONV, which states that if we can deduce a type A for an expression a, and B is β-equivalent to A, then A is a valid type for a. Why is this rule necessary? First, notice that rule APP might substitute an arbitrary term into the type of an expression, so the type of an expression is not necessarily in normal form (it might be an application, for example). Second, notice that the first premise of rule APP requires the type of f to be a Π expression. But suppose the type of f is an expression that evaluates to a Π expression, such as $(\lambda x : \star.x)(\Pi y : A.B)$! Rule CONV simply allows the necessary reduction to take place. The $=_\beta$ relation is defined in Figure 4.

[4] There is a slight awkwardness here, because you have to read the notation backwards: $p \leadsto q$ means "q can depend on p". Also confusingly, we read "terms" for "$\star$" and "types" for "$\Box$", because the $\leadsto$ judgement is two levels away from the original thing!

$$\frac{}{\vdash \star : \Box} \quad (\mathrm{STAR})$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad (\mathrm{VAR})$$

$$\frac{\Gamma \vdash f :\twoheadrightarrow (\Pi x : A.B) \qquad \Gamma \vdash a : A' \qquad A =_\beta A'}{\Gamma \vdash f\ a : B[x := a]} \quad (\mathrm{APP})$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \quad (\mathrm{LAM})$$

$$\frac{\Gamma \vdash A :\twoheadrightarrow s \qquad \Gamma, x : A \vdash B :\twoheadrightarrow t \qquad \vdash s \leadsto t}{\Gamma \vdash (\Pi x : A.B) : t} \quad (\mathrm{PI})$$

$$\frac{\Gamma \vdash a : A \qquad A \twoheadrightarrow^{wh} B}{\Gamma \vdash a :\twoheadrightarrow B} \quad (\mathrm{RED})$$

Figure 5: Syntax Directed rules for the Lambda Cube

3.7 Syntax directed rules

The inference rules given in Figure 2 are not directly suitable for use in a Core Lint type checker as the rules are not syntax directed. In particular, you cannot decide at which point in a derivation the rule CONV must be applied simply by looking at the structure of the term under consideration.

What we seek is a syntax-directed presentation of the rules that is both sound and complete with respect to the old set; that is, the new presentation should type exactly the same terms in exactly the same way. The generality of PTSs makes this task quite tricky, but Pollack, van Benthem Jutting & McKinna [1993] have done much of the hard work for us. Figure 5 gives a simplified version of their $\vdash_{\!\!\!\!\!/}$ syntax directed system. The main difference compared with the rules of Figure 2 is the strategic distribution of reduction ($\twoheadrightarrow$) over the other rules. The notation $A \twoheadrightarrow^{wh} B$ means that A reduces to the weak head normal form B. The new judgement form $\Gamma \vdash a :\twoheadrightarrow B$ means that $\Gamma \vdash a : A$ and $\Gamma \vdash A \twoheadrightarrow^{wh} B$, as rule RED states.

We have also taken the opportunity to introduce the so-called "valid context" optimization. If we assume that the initial context is well-formed (a notion it is easy to define formally), then it will remain well-formed because the only rules that extend it (LAM and PI) do so with a binding whose type is well-formed. There is therefore no need to check for this well-formedness in rules VAR and WEAK. Furthermore adopt the Barendregt convention that all variable names are distinct. As a result, we can regard Γ as an unordered bag rather than a sequence, and this means we can elininate WEAK altogether, in favour of the premise $x : A \in \Gamma$ in VAR. All of this is well known (Pollack, van Benthem Jutting & McKinna [1993, Section 2.4.]).

$\vdash_{type} a : A$ means "a has type A"

$$\frac{}{\vdash_{type} \star : \Box} \quad (\text{STAR}_{type})$$

$$\frac{}{\vdash_{type} (x : A) : A} \quad (\text{VAR}_{type})$$

$$\frac{\vdash_{type} f : F}{\vdash_{type} (f\ a) : F\ a} \quad (\text{APP}_{type})$$

$$\frac{\vdash_{type} b : B}{\vdash_{type} (\lambda x : A.b) : (\Pi x : A.B)} \quad (\text{LAM}_{type})$$

$$\frac{\vdash_{type} B : s}{\vdash_{type} (\Pi x : A.B) : s} \quad (\text{PI}_{type})$$

$$\frac{\vdash_{type} A : B \qquad B =_{\beta,\Pi} B'}{\vdash_{type} A : B'} \quad (\text{CONV}_{type})$$

Figure 6: The type-of judgement, $\vdash_{type}$.

3.8 Factoring the typing judgement $\vdash$

The typing judgement $\vdash$ actually does two things:

- It checks that an expression is well formed.
- It finds its type.

Inside a compiler one would hope that the program was always well-formed (after an initial type-check, that is), but the compiler might quite frequently want to find the type of an expression.

The "type-of" judgement, $\vdash_{type}$, in Figure 6, finds the type of a well-formed expression, *without using an environment.* To achieve this we annotate each *bound occurrence* of a variable with its type. This latter property is very useful in practice, because it means that the compiler can contain a simple function (corresponding to $\vdash_{type}$) that maps a expression to its type, without needing to plumb around an environment.

But is it not rather expensive to annotate every variable occurrence? Not necessarily. If the compiler maintains the expression as a graph, it can use a single data structure to represent `x:Int`, say, and simply point to that data structure from the binding site and each occurrence site.

The APP_{type} rule in Figure 6 also uses a neat trick due to Kamareddine and Nederpelt (Kamareddine & Nederpelt [1996]). Rather than actually perform the substitution in the rule, as we did in APP (Figure 2), we simply apply the type of the function, F, to the argument, a. As before, CONV_{type} allows us to evaluate applications when necessary, but with the additional Π-reduction rule:

$$(\Pi x : A.B)a \quad \rightarrow_{\Pi} \quad B[x := a] \qquad (\Pi)$$

This new presentation has the practical advantage in a compiler of allowing us to separate the type-finder from the evaluator in the compiler, since $\vdash_{type}$ no longer mentions substitution. Instead, we can extract the type of an expression and only then evaluate it.

A curious feature of this way of doing things is that the type of an expression may not be well formed! Consider the expression (`id Int`). Rule APP_{type} would say that it has type $(\Pi\alpha : \star.\alpha \rightarrow \alpha)$ `Int`. This type evaluates to the well-formed `Int` $\rightarrow$ `Int` as expected, but it is not itself well-formed. Why? Because $(\Pi\alpha : \star.\alpha \rightarrow \alpha)$ has type $\star$ rather than a Π type. We are a bit suspicious of this intermediate ill-formedness, but its advantages are persuasive.

Note that the rules VAR and APP rules of figure 5 can easily be modified to incorporate the changes introduced in this section.

4 A predicative variant

One disadvantage of the system we have described so far is that it is *impredicative.* In an impredicative system, type variables can range over universally-quantified types. For example, suppose that `f` : $\forall\alpha.$`[`α`]` $\rightarrow$ `Int`. Then the following type application is legitimate:

$$\texttt{f}\ (\forall\beta.\beta \rightarrow \beta)$$

Here, `f` is instantiated at the polytype $(\forall\beta.\beta \rightarrow \beta)$. There is nothing wrong with this, in the sense that "well-typed programs can't go wrong", but the ability for type variable to range over polytypes greatly complicates the business of providing a *model* for the calculus (Mitchell [1996, Chapter 9]).

4.1 Defining the predicative variant

Fortunately, it is fairly easy to produce a *predicative* variant of our calculus. Instead of just the kind $\star$ we need two constants:

$\star$	is the kind of monotypes
$\star\star$	is the kind of polytypes

Corresponding to these two kinds are two super-kinds $\Box$ and $\Box\Box$, with $\star : \Box$ and $\star\star : \Box\Box$. The latter requires a new rule, STAR2, given in Figure 7.

To make the system predicative requires that we make more distinctions about what can depend on what. This is what the new rule PI' in Figure 7 does. It makes use of a *three-place* judgement $\vdash s \rightsquigarrow t : u$, also defined in the same Figure.

The first four rules of the new $\rightsquigarrow$ judgement specify which types are polytypes and which are monotypes. For example, assuming that `Int` : $\star$ we can deduce that

$$\begin{array}{rcl}
\texttt{Int} & : & \star \\
\texttt{Int} \rightarrow \texttt{Int} & : & \star \\
\forall\alpha.\texttt{[}\alpha\texttt{]} \rightarrow \texttt{Int} & : & \star\star \\
(\forall\alpha.\texttt{[}\alpha\texttt{]} \rightarrow \texttt{Int}) \rightarrow \texttt{Int} & : & \star\star
\end{array}$$

$$\frac{}{\Gamma \vdash \star\star : \Box\Box} \quad \text{(STAR2)}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : t \quad \vdash s \rightsquigarrow t : u}{\Gamma \vdash (\Pi x : A.B) : u} \quad \text{(PI')}$$

$$\frac{}{\vdash \star \rightsquigarrow \star : \star} \quad (\star \rightsquigarrow \star)$$

$$\frac{}{\vdash \star \rightsquigarrow \star\star : \star\star} \quad (\star \rightsquigarrow \star\star)$$

$$\frac{}{\vdash \star\star \rightsquigarrow \star : \star\star} \quad (\star\star \rightsquigarrow \star)$$

$$\frac{}{\vdash \star\star \rightsquigarrow \star\star : \star\star} \quad (\star\star \rightsquigarrow \star\star)$$

$$\frac{}{\vdash \Box \rightsquigarrow \star : \star\star} \quad (\Box \rightsquigarrow \star)$$

$$\frac{}{\vdash \Box \rightsquigarrow \star\star : \star\star} \quad (\Box \rightsquigarrow \star\star)$$

$$\frac{}{\vdash \Box \rightsquigarrow \Box : \Box} \quad (\Box \rightsquigarrow \Box)$$

Figure 7: Modified rules for the predicative variant

As the last example shows, a value of polymorphic type can still be passed as an argument to a function, but the resulting function type is of kind $\star\star$. Notice that in the third rule t and u differ; that is why we now need a three-place judgement.

The next two rules in $\rightsquigarrow$, ($\Box \rightsquigarrow \star$ and $\Box \rightsquigarrow \star\star$) say that it is legitimate to abstract a monotyped variable from either a monotype or a polytype, giving a polytype in either case.

Finally, $\Box \rightsquigarrow \Box$ says that it is OK to create lambda abstractions at the type level, provided we only abstract a monotyped variable from a monotype. One could imagine also having $\Box \rightsquigarrow \Box\Box$ (allowing us to abstract a monotyped variable from a polytype), but there doesn't seem to be a pressing reason to add it, and life is simpler without.

The important thing is that there is no axiom of the form $\vdash \Box\Box \rightsquigarrow ...$, corresponding to the fact that we cannot abstract a polytyped variable from anything.

4.2 Pure Type Systems

The rules given in Figure 7 take us outside the lambda cube. Fortunately, even this generalisation has been well studied. The generalised PI' rule, together with the original rules VAR, LAM, APP, define a *Pure Type System*. A PTS is defined by:

- The rules VAR, LAM, APP and PI'.
- A set, $\mathcal{S}$, of constants ($\{\star, \star\star, \Box, \Box\Box\}$ in our case).
- A set, $\mathcal{A}$, of typing axioms relating these constants ($\{\star : \Box, \star\star : \Box\Box\}$ in our case).
- A set, $\mathcal{R}$, of $\rightsquigarrow$ rules, ranging over the constants $\mathcal{S}$II.

A PTS is called *functional* if:

- $(s, t_1) \in \mathcal{A}$ and $(s, t_2) \in \mathcal{A} \;\Rightarrow\; t_1 = t_2$
- $(s, t, r_1) \in \mathcal{R}$ and $(s, t, r_2) \in \mathcal{R} \;\Rightarrow\; r_1 = r_2$

In practice almost all practically useful PTSs, and certainly the ones in this paper, are functional. Many useful theorems (such as the substitution lemma, subject reduction) have been proved for arbitrary PTSs, and more are true of functional PTSs (such as uniqueness of types) (Barendregt [1992, Section 5.2]). Functional PTSs seem to combine these desirable theorems with a remarkable degree of flexibility — for example, we developed the predicative variant of this section well after the first draft of this paper was completed.

5 A Real Language

We now elaborate the small language of the previous section into a full language, complete with a concrete syntax. The full language[5] is given in Figure 8. Compared to the previous section we add the following features:

- A *program* consists of:
  - A set of mutually recursive `data` declarations, each of which introduces a new *data type* (Section 5.1).
  - A sequence of *value declarations*, each of which may be recursive or non-recursive (Section 5.2).
- Expressions are augmented with `let`, `letrec`, and `case` (Section 5.4).
- A special anonymous variable, "_", is provided (Section 5.3.
- A variety of abbreviations are provided (Section 5.5) as syntactic sugar. These abbreviations are marked with "†" in the left column of Figure 8. Their purpose is to reduce the number of characters it takes to print out a program, and make it more comprehensible to the human reader.

There is only one name space. Haskell programs that use the same name for a data type and a data constructor will need to be renamed before being expressed in Henk.

[5] Actually there is still one production to come, for primitive operations.

Program	*program*	$\rightarrow$	$tdecl_1 \; \ldots \; tdecl_n \; vdecl_1 \; \ldots \; vdecl_m$	$(n \geq 0, m \geq 0)$
Type declaration	*tdecl*	$\rightarrow$	`data` *tvar* `= {` $tvar_1 \; \ldots \; tvar_n$ `}`	$(n \geq 1)$
Value declaration	*vdecl*	$\rightarrow$	`let {` *bind* `}`	Non-recursive
		$\mid$	`letrec {` $bind_1 \; \ldots \; bind_n$ `}`	Recursive $(n \geq 1)$
Binding	*bind*	$\rightarrow$	*tvar* `=` *expr*	
Expression	*expr*	$\rightarrow$	*bexpr*	
		$\mid$	`\` $tvar_1 \; \ldots \; tvar_n$ `.` *expr*	λ $(n \geq 1)$
		$\mid$	`|~|` $tvar_1 \; \ldots \; tvar_n$ `.` *expr*	Π $(n \geq 1)$
†		$\mid$	`/\` $tvar_1 \; \ldots \; tvar_n$ `.` *expr*	Λ $(n \geq 1)$
†		$\mid$	`\/` $tvar_1 \; \ldots \; tvar_n$ `.` *expr*	$\forall$ $(n \geq 1)$
†		$\mid$	*bexpr* `->` *expr*	$\rightarrow$
		$\mid$	*vdecl* `in` *expr*	Local declaration
		$\mid$	`case` *expr* `of {` alt_1 `;` $\ldots$ `;` alt_n `}`	$(n \geq 1)$
			`at {` $aexpr_1 \; \ldots \; aexpr_m$ `}`	$(m \geq 0)$
†		$\mid$	`case` *expr* `of {` alt_1 `;` $\ldots$ `;` alt_n `}`	
	bexpr	$\rightarrow$	*bexpr aexpr*	Application
		$\mid$	*aexpr*	
	aexpr	$\rightarrow$	*tvar*	Variable
		$\mid$	*literal*	Literal
		$\mid$	`*`	Constant
		$\mid$	`BOX`	Constant
		$\mid$	`(` *expr* `)`	
Typed variable	*tvar*	$\rightarrow$	*var* `:` *aexpr*	
†		$\mid$	*var*	
Variable	*var*	$\rightarrow$	`_`	(binding sites only)
		$\mid$	*identifier*	
Case alternative	*alt*	$\rightarrow$	*pat* `->` *expr*	
†		$\mid$	*pat* $tvar_1 \; \ldots \; tvar_n$ `->` *expr*	
Case patterns	*pat*	$\rightarrow$	*tvar*	
		$\mid$	*literal*	

"†" indicates syntactic sugar

Figure 8: Concrete syntax

5.1 Type declarations

An important design choice is that we introduce new types with explicit declarations, outside the syntax of expressions. A data declaration defines a new algebraic data type. It introduces a new *type constructor* plus a number of *data constructors* into the environment. For example the type of generalized trees with values of type `a` is defined as:

```
data Tree : ((*->*) -> * -> *)
= { Branch : (\/ (m:*->*) (a:*) .
               a -> m (Tree m a) -> Tree m a)
  }
```

This declaration introduces the type constructor `Tree` and the data constructor `Branch`, each with the specified type.

Like ML, and unlike Haskell, type constructors and data constructors are not required to start with an upper-case letter.

The main design alternative would be to provide primitive type constructors for unit, sum, product, lifting, and recursion, and then use ordinary value declarations to introduce new types. For example, we might introduce lists thus:

```
List:(*->*) = \a:*. rec (\l:*. Lift (Unit + (a ** l)))
```

Whilst this is nice from a theoretical standpoint, it has several disadvantages:

- It does not generalise easily to handle mutual recursion and non-uniform data types; the former is very common, and the latter legal, in both ML and Haskell.
- It is hard to know when to unwind the recursion, for example when testing types for equality.
- It is harder to prove strong normalisation for types.

We have opted to be conservative and exploit the type structure we know we have. The extra generality of arbitrary products and sums does not seem worth the complexity.

5.2 Value Declarations

A value binding binds a variable to a value. This can either be a term value (e.g. `x:Int = 3`), or a type value (e.g. `Diag:(*->*) = \a.Pair a a`). Value bindings can be grouped into recursive or non-recursive declarations. These value declarations appear both at the top level of a program, and as local declarations inside expressions.

5.3 The anonymous variable, _

The anonymous variable, `_`, can be used at the binding site of a variable that is not mentioned in its scope. It is useful in the concrete syntax to reduce the creation of new names. More importantly, it is useful inside the compiler because it allows the evaluator to eliminate a substitution step when applying an abstraction (λ or Π) whose bound variable is unused. Such abstractions are very common indeed: every function arrow turns into one!

5.4 Case expressions

A `case` expression takes apart values built with constructors. Here is an example:

```
case (reverse xs) of
   { Cons -> \y ys . <rhs1>
   ; Nil  -> <rhs2>
   }
at { Int }
```

A `case` expression scrutinises an expression, called the *scrutinee*. The scrutinee is `reverse xs` in this example. It evaluates the expression to head normal form, and matches it against the alternatives. The pattern in a case alternative must be a literal, a constructor name, or `_`. All the patterns in a `case` expression must have the same type. The list of types in the `at` clause gives the types at which the constructors are instantiated (it is empty if the patterns are literals). In the example, `xs` is presumably a list of `Int`, so `Cons` and `Nil` are instantiated at `Int`.

When an alternative is selected, its right-hand-side is applied to the values of the arguments of the constructor. Thus, if `reverse xs` evaluates to Cons Int <e1> <e2>, the result of the `case` expression will be:

```
(\y ys . <rhs1>) <e1> <e2>
```

If `_` is used as a pattern, it is selected only if all the others fail to match, regardless of order.

This form of `case` is a little unusual. More typically the patterns can also bind variables, thus:

```
case (reverse as) of
  { Cons y ys -> <rhs1>
  ; Nil       -> <rhs2>
  }
```

We provide the conventional form as syntactic sugar (Section 5.5), but the core form reduces the number of expressions that bind new variables. This in turn reduces clutter in the compiler, without losing expressiveness.

5.5 Syntactic sugar

The following syntactic sugar greatly reduces the size of the printed form of a program (apart from the first two, which simply give more conventional equivalent forms):

- `/\` means the same as `\`.
- `\/` means the same as `|~|`.
- $\backslash x_1 \ldots x_n . e$ means the same as $\backslash x_1 . \ldots \backslash x_n . e$.
- e_1 `->` e_2 means the same as `|~| _:`e_1`.` e_2.
- A *binding* occurrence of an un-annotated variable v means the same as v`:*`. This allows us to omit the annotation for most type variables.
- A *bound* occurrence of an un-annotated variable v means the same as v:t, where t is the annotation at its binding site. (It may be necessary to perform

α-conversion for this to have the expected meaning.) This abbreviation allows us to omit annotations on all bound occurrences.

- The `at` clause on a `case` expression can be omitted, because it can readily be re-inferred.
- The case alternative $c\ y_1 \dots y_n$ `->` e means the same as c `->` `\`$y_1 \dots y_n$ `.` e. The variables $y_1, \dots, y_n$ need not be annotated with their types; if they are not, their types are recovered from the type of the constructor and the instantiating types in the `at` clause.

5.6 Type rules

Figure 9 extend our typechecking rules to deal with the extended language.

The $\vdash$ rules for declarations return an environment as the "type" of a declaration. To avoid clutter, the rules for `letrec` and `case` mention only a single binding or alternative respectively.

The main interesting point is in the LET and CASE rules. Their form is very like the APP rule in Figure 6, in that the type derived is of the same form as the expression being typed. We rely on the conversion rules (which we don't have space for here) to convert the resulting type to the required form where necessary (i.e. in APP and CASE).

Rule LET raises an interesting question. Clearly, we need check that it is legal to abstract the bound variable(s) over the body ($\vdash \Delta \leadsto A$). But, if the binding is recursive — say, `letrec {`x_1 `=` a_1 `;` x_2 `=` a_2`}` — do we need to check that it is legal to abstract the bound variables over each of the a_i? We believe that the answer is no. To see why, consider the expression `let {`$x : A$ `=` a`} let {`$y : B$ `=` b`}` c. Here, we clearly do not check that we can abstract $x : A$ over b. It is not clear to us what the correct answer is here, but we plan to find out!

6 Conclusions and further work

We believe that the lambda cube provides a solid foundation for the intermediate langauge of sophisticated compilers. It is a subtle system, and extending it to a real language raises new technical issues, as we have just seen. Probably some of these problems are old hat to the theorists, which is why the direct link to a well-studied system is so valuable.

There is still much groundwork left to do:

- We have gaily extended the PTS framework with recursive data types, `let`, `letrec`, `case`, and constants, but it is necessary to prove that all the standard PTS theorems still go through. (Indeed, we noted some uncertainty about the exact typing rule for `letrec` in the previous section.)
- Section 4 defines a predicative variant of Henk, with the intention that it has a more tractable model; but we have yet to exhibit such a model.
- We should provide an operational semantics for Henk.

Subsequently, we plan to move towards an implementation in GHC. More ambitiously, we hope that by giving Henk a clear definition, and a concrete syntax, we may be able to work more closely with other compiler-writers. GHC has always been intended as a substrate for others' research, but it is such a daunting monster that it requires a certain determination to delve into its innards. We hope that Henk may provide a route from GHC into other back ends or analysis tools, and vice versa.

7 Acknowledgements

Thanks for Koen Claessen, John Launchbury, Randy Pollack, and Mark Shields for comments on earlier drafts. We acknowledge gratefully the support of the Oregon Graduate Institute during our sabbaticals, funded by a contract with US Air Force Material Command (F19628-93-C-0069).

References

H Barendregt [1992], "Lambda calculi with types," in *Handbook of Logic in Computer Science, Volume II*, S Abramsky, DM Gabbay & TSE Maibaum, eds., Oxford University Press.

TH Coquand & G Huet [1988], "The calculus of constructions," *Information and Computation* 76, .

A Gill, J Launchbury & SL Peyton Jones [June 1993], "A short cut to deforestation," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, .

J-Y Girard [1972], "Interprétation foctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur," PhD thesis, Université Paris VII.

R Harper & G Morrisett [Jan 1995], "Compiling polymorphism using intensional type analysis," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, .

RW Harper, JC Mitchell & E Moggi [Jan 1990], "Higher-order modules and the phase distinction," in *17th ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, .

Z Hu, H Iwasaki & M Takeichi [May 1996], "Deriving structural hylomorphisms from recursive definitions," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, .

MP Jones [Jan 1995], "A system of constructor classes: overloading and implicit higher-order polymorphism," *Journal of Functional Programming* 5, .

$$\frac{\begin{array}{lll}\Gamma \cup \bigcup_{1\leq i\leq n} \Delta_i & \vdash & t_j : \Delta_k \quad (1 \leq k \leq n) \\ \Gamma \cup \bigcup_{1\leq i\leq n} \Delta_i \cup \bigcup_{1\leq j\leq k-1} \Theta_j & \vdash & v_k : \Theta_k \quad (1 \leq k \leq m)\end{array}}{\Gamma \vdash t_1 \ldots t_n \; v_1 \ldots v_n} \quad \text{(PROG)}$$

$$\frac{\Gamma \vdash a : A}{\Gamma, x : A \vdash \texttt{let } \{x : A = a\} : \{x : A\}} \quad \text{(VDECL1)}$$

$$\frac{\Gamma, x : A \vdash a : A}{\Gamma, x : A \vdash \texttt{letrec } \{x : A = a\} : \{x : A\}} \quad \text{(VDECL2)}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : t}{\Gamma, x : A, y : B \vdash \texttt{data } x : A = \{ \; y : B \; \} : \{x : A, y : B\}} \quad \text{(DATA)}$$

$$\frac{\Gamma \vdash vdecl : \Delta \quad \Gamma, \Delta \vdash a : A \quad \vdash \Delta \leadsto A}{\Gamma \vdash vdecl \texttt{ in } a : vdecl \texttt{ in } A} \quad \text{(LET)}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : X \to B \quad \Gamma \vdash (c \; T) : X \to A}{\Gamma \vdash_{\mathit{wff}} \texttt{case } a \texttt{ of } \{ \; c \texttt{ -> } b\} \texttt{ at } \{ \; T \; \} : \texttt{case } a \texttt{ of } \{ \; c \texttt{ -> } B\} \texttt{ at } \{ \; T \; \}} \quad \text{(CASE)}$$

Figure 9: Extra rules for $\vdash$

F Kamareddine & R Nederpelt [March 1996], "Canonical typing and Pi-conversion in the Barendregt Cube," *Journal of Functional Programming* 6, .

J Launchbury & T Sheard [June 1995], "Warm fusion," in *Proc Functional Programming Languages and Computer Architecture, La Jolla*, ACM.

JC Mitchell [1996], *Foundations for programming languages*, MIT Press.

G Morrisset [Dec 1995], "Compiling with types," PhD thesis, CMU-CS-95-226, Carnegie Mellon University.

SL Peyton Jones [Jan 1996], "Compilation by transformation: a report from the trenches," in *European Symposium on Programming (ESOP'96), Linköping, Sweden*, Springer Verlag LNCS 1058, .

SL Peyton Jones, CV Hall, K Hammond, WD Partain & PL Wadler [March 1993], "The Glasgow Haskell compiler: a technical overview," in *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, DTI/SERC, .

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, .

SL Peyton Jones & WD Partain [1993], "Measuring the effectiveness of a simple strictness analyser," in *Functional Programming, Glasgow 1993*, K Hammond & JT O'Donnell, eds., Workshops in Computing, Springer Verlag, .

R Pollack, L van Benthem Jutting & J McKinna [May 1993], "Typechecking in pure type systems," in *Types for Proofs and Programs, Nijmegen, May '93, Selected Papers*, Springer Verlag LNCS 806.

Z Shao [1996], "The FLINT compiler system," Presentation at IFIP Working Group 2.8.

Z Shao & AW Appel [June 1995], "A type-based compiler for Standard ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'95), La Jolla*, ACM, .

D Tarditi [1996], "Optimizing ML," PhD thesis, Carnegie Mellon University.

D Tarditi, G Morrisett, P Cheng, C Stone, R Harper & P Lee [May 1996], "TIL: A Type-Directed Optimizing Compiler for ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96), Philadelphia*, ACM.

A Tolmach [June 1994], "Tag-free garbage collection using explicit type parameters," in *ACM Symposium on Lisp and Functional Programming, Orlando*, ACM, .