A Type-Preserving Compiler in Haskell

Louis-Julien Guillemette Stefan Monnier
Université de Montréal
{guillelj,monnier}@iro.umontreal.ca

Abstract

There has been a lot of interest of late for programming languages that incorporate features from dependent type systems and proof assistants, in order to capture important invariants of the program in the types. This allows type-based program verification and is a promising compromise between plain old types and full blown Hoare logic proofs. The introduction of GADTs in GHC (and more recently type families) made such dependent typing available in an industry-quality implementation, making it possible to consider its use in large scale programs.

We have undertaken the construction of a complete compiler for System F, whose main property is that the GHC type checker verifies mechanically that each phase of the compiler properly preserves types. Our particular focus is on "types rather than proofs": reasonably few annotations that do not overwhelm the actual code.

We believe it should be possible to write such a type-preserving compiler with an amount of extra code comparable to what is necessary for typical typed intermediate languages, but with the advantage of static checking. We will show in this paper the remaining hurdles to reach this goal.

Categories and Subject Descriptors D.3 [*Software*]: Programming languages; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanic al verification

General Terms Languages, Verification

Keywords Compilation, Typed assembly language, de Bruijn, Higher-Order Abstract Syntax

1. Introduction

Formal methods are rapidly improving and gaining ground in software, and type systems are arguably the most successful and popular formal method used to develop software. As the technology of type systems progresses, new needs and new opportunities appear. One of those needs is to ensure the faithfulness of the translation from source code to machine code, so that the properties you prove about the code you write also apply to the code you run.

Recent work on certified compilation makes successful use of proof assistants based on dependent types to establish semantic preservation. Leroy's compiler (Leroy 2006; Blazy et al. 2006) proves dynamic semantic preservation for a first-order (C-like) language. Chlipala's compiler (2007) uses other techniques to provide similar guaranties for a functional language. Both are developed as Coq proofs from which a working compiler is obtained by means of program extraction.

With the introduction of *generalized algebraic data types* (GADTs) in the Glasgow Haskell Compiler (GHC), and more recently *type families* (Schrijvers et al. 2007), a useful (if limited) form of dependent typing is finally available in an industry-quality implementation of a general-purpose programming language. Thus arises the possibility of establishing compiler correctness through type annotations in Haskell code, without the need to encode elaborate proofs as separate artifacts. In this work, we use types to enforce *type preservation*: our typed intermediate representation lets GHC's type checker manipulate and check our object types.

Other than the CPS conversion over System F of Chlipala (2008) developed in parallel and presented elsewhere in these proceedings, previous work invariably restricts the input to simply typed features. Ours handles the full System F extended with recursive definitions; we are able to cover a larger set of object language features by focusing on type preservation rather than full correctness. Extensions like first-class existential and recursive types (which seem well within reach at this point) would enable the encoding of algebraic datatypes, and our compiler could serve as a back-end for Core ML and similar languages.

By using Haskell rather than a language with full dependent types, we narrow down the semantic gap between the host and object language, bringing closer the possibility of a bootstrapping type-preserving compiler. Our implementation relies essentially on GADTs and type families – and all these features can be encoded in a variant of System F with type equality coercions (Sulzmann et al. 2007).

Typed intermediate languages have been widely used in compilers, often as a manner of sanity check to help catch compiler errors. Typically object types are represented in the form of data structures which have to be carefully manipulated to be kept in sync with the code they annotate as this code is being transformed. In comparison, our approach enforces type preservation statically through type annotations in the compiler's code, with obvious advantages: it is exhaustive, unlike the conventional approach which amounts to *testing* the compiler; it gives earlier detection of errors in the compiler; it also eliminates the overhead of manipulating and checking type annotations while running the compiler. Last but not least, it can actually simplify the compiler's code, as no explicit manipulations of type information are needed.

Our main contributions are the following:

- We show a CPS and closure conversion over System F where type preservation is enforced by type families and type equality constraints. This constitutes the first mechanized argument of type preservation for these transformations over System F.
- We extend the classical toy example of a GADT representation of an abstract syntax tree, to a full language with term-level and type-level bindings. We show both a higher-order and a first-

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

order encoding of System F and address subtle issues about the interaction of bindings at the levels of types and terms.

- To our knowledge, this is the first worked out example of extensive use of GHC type families and type equality coercions. Thereby it feeds the current debate as to which one of type families, associated types, or multiple-parameter type classes with functional dependencies, should make it to the next Haskell standard (Peyton-Jones et al. 2007).
- We argue that none of the existing representations of bindings is suitable in the sense that they either cannot be used, or they introduce significant extra complexity.

The paper is structured as follows. After a brief overview of the compiler and the techniques it employs (Sec. 2), we present the encoding of System F (Sec. 3) and review the implementation of the individual compilation phases (Sec. 4 through 7). We finally discuss our experience in general terms (Sec. 8) and mention related work (Sec. 9).

2. Overview and background

This section introduces the types and techniques we use to make the compiler type-preserving, and describes the overall structure of the compiler.

2.1 Generalized algebraic datatypes

The program representations we use are constructed with Generalized Algebraic Datatypes, or GADTs (Xi et al. 2003; Cheney and Hinze 2003). They are a generalization of algebraic datatypes where the return types of the various data constructors for a given datatype need not be identical – they can differ in the type arguments given to the type constructor being defined. The type arguments can be used to encode additional information about the value that is represented. For our purpose, we use these type annotations to track the object-level type of expressions. For example, consider some common typing rules:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Using plain algebraic datatypes, we would represent object programs with a type such as the following:

```
data Exp where
  Enum :: Int → Exp
  Eadd :: Exp → Exp → Exp
  Eapp :: Exp → Exp → Exp
```

where source types (that is, the types in the object program) of e_1 and e_2 are unconstrained. In contrast, with GADTs, we can explicitly mention source types as type arguments to *Exp* to encode the three typing rules:

```
data Exp t where
  Enum :: Int → Exp Int
  Eadd :: Exp Int → Exp Int → Exp Int
  Eapp :: Exp (t1 → t2) → Exp t1 → Exp t2
```

This type guarantees that if we can construct a Haskell term of type *Exp* t, then the source expression it represents is well typed: it has some type τ, the source type for which t stands. Note that the use of the arrow constructor $(t_1 \to t_2)$ to represent object-level function types $(\tau_1 \to \tau_2)$ is purely arbitrary: we could just as well have used any other type of our liking, say *Arw* $t_1\ t_2$, to achieve the same effect.

2.2 Binders

There are different possible ways to extend the GADT *Exp* to encode syntactic constructs that involve binders, depending on whether variables are represented explicitly or implicitly using the host language's variables. Either representations has its advantages and limitations, and we use each in different parts of the compiler. Using both representations within one compiler may seem odd (and it is!), and reflects a course of experimentation more than an engineering decision.

The representation of binders at the *type* level falls beyond the scope of background material, so we describe it later. We show how we do it in the context of a higher-order term representation when describing the encoding of the source language (Sec. 3), and in the context of a first-order one when dealing with closure conversion (Sec. 6). We discuss further alternatives in Sec. 8.

HOAS. Consider the usual typing rule for let-expressions:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

With Higher-Order Abstract Syntax, we would encode the typing rule as follows:

```
data Exp t where
  Elet :: Exp t1 → (Exp t1 → Exp t2) → Exp t2
  ...
```

that is, binders in source programs would be represented by Haskell binders – and thus variable occurrences would not require an explicit introduction form. As long as bindings in the source language behave the same as bindings in Haskell, the technique amounts to re-using Haskell's (implicit) type contexts to impose type constraints on source programs.

De Bruijn indices. In contrast to HOAS, a first-order representation introduces variables explicitly. With de Bruijn indices, as with HOAS, variables's names are irrelevant, and variables are instead represented as indices. The type associated with an index is drawn from an explicit type argument (*ts*) to *Exp*, which represents the expression's type context:

```
data Exp ts t where
  Bnum :: Int → Exp ts Int
  Badd :: Exp ts Int → Exp ts Int → Exp ts Int
  Bapp :: Exp ts (t1 → t2) → Exp ts t1 → Exp ts t2

  Bvar :: Index ts t → Exp ts t
  Blet :: Exp ts s → Exp (s, ts) t → Exp ts t
```

A term of type *Exp ts t* is an expression that may refer to variables whose types are listed in *ts*. More precisely, a Haskell term being of type *Exp ts t* implies that the source term it represents (e) satisfies $\Gamma \vdash e : \tau$, where the Haskell type t stands for the source type τ, and the type *ts* reflects Γ.

An index of type *Index ts t* represents a de Bruijn index whose type is t within the type environment *ts*. Such indices are represented with type-annotated Peano numbers:

```
data Index ts t where
  I0 :: Index (t, ts) t
  Is :: Index ts t → Index (t0, ts) t
```

Note that individual indices are polymorphic in *ts* and t, and assume a particular type given a particular type context *ts*. (Specifically, for an index of the form $Is^i\ I0$ of type *Index ts t*, the only relation between *ts* and t is that the i^{th} type appearing in *ts* is t, which is why t_0 appears free in the type of *Is*, and *ts* appears free in that of *I0*.)

To illustrate the two techniques, the following expression:

$$\begin{array}{l}\mathsf{let}\ a = 2\\ \quad\ \ b = 3\\ \mathsf{in}\ a + b\end{array}$$

would be represented in HOAS as:

Elet (*Enum* 2) ($\lambda a \rightarrow$
  Elet (*Enum* 3) ($\lambda b \rightarrow$
    Eadd a b))

and with de Bruijn indices as:

Blet (*Bnum* 2) (
  Blet (*Bnum* 3) (
    Badd (*Bvar* (*Is* *I*0)) (*Bvar* *I*0)))

2.3 Type families

Type families (Schrijvers et al. 2007) are a recent addition to GHC that allows programmers to directly define functions over types by case analysis, in a way that resembles term-level function definitions with pattern matching.

For example, we can define a type function *Add* that computes (statically) the sum of two Peano numbers:

data Z; **data** $S\ i$ — natural numbers encoded as types

type family *Add* n m
type instance *Add* Z $m = m$
type instance *Add* (S n) $m = S$ (*Add* n m)

We can then use this type family to express the fact that an *append* function over length-annotated lists produces a list of the expected length:

data *List elem len* **where**
  Cons :: *elem* $\rightarrow$ *List elem* $n \rightarrow$ *List elem* (S n)
  Nil :: *List elem* Z

append :: *List elem* $n \rightarrow$ *List elem* $m \rightarrow$ *List elem* (*Add* n m)
append Nil $l = l$
append (*Cons* h t) l = *Cons* h (*append* t l)

Note that the definition of *append* follows the structure of the definition of the type function *Add*, so that the type checker can verify that every clause of *append* satisfies its type signature.

We mention in passing that GHC's implementation of type families relies on the calculus of type equality coercions (Sulzmann et al. 2007), and that these coercions themselves are exposed to the programmer, a feature that plays an important technical role here. As it is not essential to understand these at this point, we will introduce them when needed (cf. Sec. 4.3).

2.4 Compilation phases

The source language we are compiling is a call-by-value functional language with parametric polymorphism and recursion, similar to System F. The general compilation strategy follows that of Morrisett et al. (1999). The overall structure of our compiler is as follows:

$$\lambda \xrightarrow{\text{typecheck}} \lambda_{\rightarrow} \xrightarrow[convert]{\text{CPS}} \lambda_{\mathcal{K}} \xrightarrow[convert]{deBruijn} \lambda^{b}_{\mathcal{K}} \xrightarrow[convert]{closure} \lambda^{b}_{\mathcal{C}} \xrightarrow{\text{hoist}} \lambda^{b}_{\mathcal{H}}$$

The source language and each intermediate language ($\lambda_{\rightarrow}$, $\lambda_{\mathcal{K}}$, etc.) has its own syntax and type system, so each is encoded as a separate GADT. The first phase infers types for all subterms of the source program, and all the subsequent ones are then careful to preserve them. In general, the way a transformation affects the types is captured by a type function.

In this section we will briefly show the effect of each phase on the code and sketch its type.

Type checking: $\boxed{AST \rightarrow \exists t.Exp\ t}$

The type checking phase takes a simple abstract data type *AST*, then it infers and checks its type t, and returns a *generalized algebraic datatype* (GADT) of type *Exp* t which does not just represent the syntax any more but a proof that the expression is properly typed, in the form of a type derivation. In order for the CPS phase to more closely match the natural presentation, we make it work on a *higher-order abstract syntax* (HOAS) representation of the code, so the type checking phase also converts the first order abstract syntax (where variables are represented by their names) to a HOAS (where variables are represented by meta variables) at the same time.

The conversion to HOAS is implemented using Template Haskell (Sheard and Jones 2002), a compile-time meta-programming facility bundled with GHC – that is, it allows us to construct a piece of Haskell code under program control. This piece of code gets type-checked by GHC, and since the program representation we construct is strongly typed, we get a source-level type checker for free. Constructing HOAS terms by meta-programming gives us an efficient representation, in contrast to a direct implementation which would lead to residual redexes (i.e. recursive calls to the conversion function hidden inside closures for functional arguments, like those for λ or let.)

CPS *conversion:* $\boxed{Exp\ t \rightarrow ExpK\ (cps\ t)}$

Conversion to *continuation-passing style* (CPS) names all intermediate results and makes the control structure of a program explicit. In CPS, a function does not return a value to the caller, but instead communicates its result by calling a *continuation*, which is a function that represents the "rest of the program", that is, the context of the computation that will consume the value produced. Additionally a special form halt is used to indicate the final "answer" produced by the program. For example:

$$\begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad\ \ b = 32\\ \quad\ \ c = 24\\ \quad\ \ c2f = \lambda x\ .\ a \cdot x\ +\ b\\ \mathsf{in}\ c2f\ c\end{array} \xrightarrow{\text{CPS}} \begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad\ \ b = 32\\ \quad\ \ c = 24\\ \quad\ \ c2f = \lambda\langle x, k\rangle\ .\\ \qquad\qquad \mathsf{let}\ v_0 = a \cdot x\\ \qquad\qquad\quad\ \ v_1 = v_0 + b\\ \qquad\qquad \mathsf{in}\ k\ v_1\\ \mathsf{in}\ c2f\ \langle c, \lambda v\ .\ \mathsf{halt}\ v\rangle\end{array}$$

For an input expression of type *Exp* t the output type should be *ExpK* (*cps* t) where *cps* is a type function that describes the way types are modified by this phase: mostly input types of the form $\tau_1 \rightarrow \tau_2$ are mapped to $\langle \tau_1', \tau_2' \rightarrow 0\rangle \rightarrow 0$ where 0 is the void type. The type of the CPS conversion (*Exp* $t \rightarrow$ *ExpK* (*cps* t)) expresses and enforces directly that the function preserves types.

Conversion to de Bruijn: $\boxed{ExpK\ t \rightarrow ExpKB\ ts\ t}$

While HOAS is convenient for the CPS conversion, it is (at least) impractical in the closure conversion, so we switch representation mid-course from $\lambda_{\mathcal{K}}$ to $\lambda^{b}_{\mathcal{K}}$ where the only difference is the representation of variables, which uses de Bruijn indices. Among other things this forces us to make the type environment explicit in the type of our terms. So for an input expression of type *ExpK* t, meaning the represented expression has type t in the current context, the return value will have type *ExpKB* ts t, which means it represent an expression of type t but this time in a type environment ts. Making the type environment explicit is crucial when we need to express the fact that a particular expression is closed, which is the key property guaranteed by the closure conversion and used by the hoisting phase.

Closure conversion: $\boxed{\textit{ExpKB ts t} \to \textit{ExpC (map cc ts) (cc t)}}$

Closure conversion makes the creation of closures explicit. Functions are made to take an additional argument, the *environment*, that captures the value of its free variables. A closure consists of the function itself, which is closed, along with a copy of the free variables forming its environment. At the call site, the closure must be taken apart into its function and environment components and the call is made by passing the environment as an additional argument to the function. For example, the above CPS example code will be transformed by the closure conversion into the following code:

$$
\begin{array}{l}
\mathsf{let}\ a = 1.8 \\
\quad b = 32 \\
\quad c = 24 \\
\quad c2f = \mathsf{pack}\ (\langle \mathsf{int}, \mathsf{int} \rangle, \\
\qquad\qquad \langle \lambda \langle \langle x, k \rangle, env \rangle\ .\ \mathsf{let}\ v_0 = env.0 \cdot x \\
\qquad\qquad\qquad\qquad\qquad\quad v_1 = v_0 + env.1 \\
\qquad\qquad\qquad\qquad\qquad\quad (\beta, \langle k_f, k_{env} \rangle) = \mathsf{unpack}\ k \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ k_f\ \langle v_1, k_{env} \rangle, \\
\qquad\qquad\quad \langle a, b \rangle \rangle) \\
\qquad\quad \mathsf{as}\ \tau_{c2f} \\
\quad (\beta, \langle c2f_f, c2f_{env} \rangle) = \mathsf{unpack}\ c2f \\
\mathsf{in}\ c2f_f\ \langle \langle c, \mathsf{pack}\ (\langle\rangle, \langle \lambda \langle v, env \rangle\ .\ \mathsf{halt}\ v, \langle\rangle \rangle)\ \mathsf{as}\ \tau_{halt} \rangle, c2f_{env} \rangle
\end{array}
$$

where $\tau_{halt} = \exists\alpha.\langle\langle \mathsf{int}, \alpha\rangle \to 0, \alpha\rangle$
$\tau_{c2f} = \exists\beta.\langle\langle\langle \mathsf{int}, \tau_{halt}\rangle, \beta\rangle \to 0, \beta\rangle$

The closure conversion takes and expression of type *ExpKB ts t* to and expression of type *ExpC* (*map cc ts*) (*cc t*). Mostly continuations ($\tau \to 0$) are mapped to closure types: $\exists\beta.\langle\langle\tau', \beta\rangle \to 0, \beta\rangle$ where the existential variable β abstracts the type of the environment, so that two functions that are of the same type before the conversion are mapped to closures of the same type, irrespective of the type of their free variables. The form pack constructs an existential package of a specified type, and unpack opens up a package, bringing in scope a type variable that stands for the abstracted type.

Currently, existential types are introduced only at the point of doing closure conversion, but we intend to add existential types to our source language in the future.

Hoisting: $\boxed{\textit{ExpC ts t} \to \textit{ExpH ts t}}$

After closure conversion, λ-abstractions are closed and can be moved to the top level. The previous example after hoisting will look as follows:

$$
\begin{array}{l}
\mathsf{let}\ \ell_0 = \lambda \langle \langle x, k \rangle, env \rangle\ .\ \mathsf{let}\ v_0 = env.0 \cdot x \\
\qquad\qquad\qquad\qquad\qquad\quad v_1 = v_0 + env.1 \\
\qquad\qquad\qquad\qquad\qquad\quad (\beta, \langle k_f, k_{env} \rangle) = \mathsf{unpack}\ k \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ k_f\ \langle v_1, k_{env} \rangle \\
\quad \ell_1 = \lambda \langle v, env \rangle\ .\ \mathsf{halt}\ v \\
\quad a = 1.8 \\
\quad b = 32 \\
\quad c = 24 \\
\quad c2f = \mathsf{pack}\ (\langle \mathsf{int}, \mathsf{int} \rangle, \langle \ell_0, \langle a, b \rangle \rangle)\ \mathsf{as}\ \tau_{c2f} \\
\quad (\beta, \langle c2f_f, c2f_{env} \rangle) = \mathsf{unpack}\ c2f \\
\mathsf{in}\ c2f_f\ \langle \langle c, \mathsf{pack}\ (\langle\rangle, \langle \ell_1, \langle\rangle \rangle)\ \mathsf{as}\ \tau_{halt} \rangle, c2f_{env} \rangle
\end{array}
$$

where τ_{halt} and τ_{c2f} are the same as above. This phase re-arranges the bindings so that all functions appear at the top level, but does not otherwise affect the types.

3. Encoding System F

This section describes the strongly typed representation of the variant of System F that constitutes our source language ($\lambda_{\to}$, whose syntax[1] is shown in Fig. 1.) An important part of the encoding is the choice of representation to use for each binding. In a language like System F there are three distinct classes of binders to consider:

1. at the term level, those that bind values (such as fix and let);
2. at the term level, those that introduce free type variables (Λ);
3. those that bind types at the type level (the $\forall$ quantifiers).

The two different kinds of type binders can be treated uniformly, but can also (to a certain extent) be handled separately, as is done in *locally nameless* representations.

[1] We abbreviate $\mathsf{fix}\ f\ x.\ e$ as $\lambda x\ .\ e$ when f does not appear free in e.

$$
\begin{array}{ll}
(\textit{types}) & \tau ::= \tau_1 \to \tau_2 \mid \forall\alpha.\ \tau \mid \alpha \mid \langle \tau_0, ..., \tau_{n-1} \rangle \mid \mathsf{int} \\
(\textit{exps}) & e ::= \mathsf{fix}\ f\ x.\ e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
 & \quad \mid x \mid e_1\ e_2 \mid \Lambda\alpha.\ e \mid e[\tau] \mid \langle e_0, ..., e_{n-1} \rangle \mid e.i \\
 & \quad \mid n \mid e_1\ p\ e_2 \mid \mathsf{if0}\ e_1\ e_2\ e_3 \\
(\textit{primops}) & p ::= + \mid - \mid \cdot
\end{array}
$$

Figure 1. Syntax of $\lambda_{\to}$

$\lambda_{\to}$ type	Haskell type
$\tau_1 \to \tau_2$	$t_1 \to t_2$
$\forall\alpha.\ \tau$	*All t*
α	*Var i*
$\langle \tau_0, ..., \tau_{n-1} \rangle$	$(t_0, (..., (t_{n-1}, ())...))$
int	*Int*

Figure 2. Encoding of $\lambda_{\to}$ types

3.1 Types

Of course, encoding System F in a GADT implies that introduction and elimination of type variables take place at Haskell's type level. While HOAS would be our preferred choice to represent type-level bindings, GHC does not provide λ-expressions at the level of types, which constrains our representation of System F types to be first-order: bound type variables are represented with type-level de Bruijn indices. The encoding of types we use is summarized in Fig. 2.

To illustrate, the type of the usual $flip$ function for pairs:

$$\forall\alpha\ \beta.\ \langle \alpha, \beta \rangle \to \langle \beta, \alpha \rangle$$

is represented as the Haskell type:

$$\textit{All}\ (\textit{All}\ ((\textit{Var}\ (S\ Z), \textit{Var}\ Z) \to (\textit{Var}\ Z, \textit{Var}\ (S\ Z))))$$

As usual the typing rule for type application eliminates a universal quantifier by applying a substitution:

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ \tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\ \tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

where Δ is a component of the type context that tracks the type variables that are in scope. If we transpose $\lambda_{\to}$ types in de Bruijn, and thus eliminate all type variable names, the typing rules become:

$$\frac{\Delta + 1; \mathsf{shift}\ \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda e : \forall\tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\tau_1 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/0]}$$

Here, Δ only tracks the *number* of type variables in scope. When extending the context (to $\Delta + 1$), the type of the term variables (in Γ) must be adjusted: all free variables must be incremented to account for the intervening binder, hence the shift operator.

The form $\tau[\tau'/i]$ yields the type τ where the type variable i has been replaced by the type τ'; it is defined in Fig. 3. It is a conventional substitution over de Bruijn terms (as in, e.g. Kamareddine

$$
\begin{array}{rl}
(\tau_1 \to \tau_2)[\tau/i] & = \tau_1[\tau/i] \to \tau_2[\tau/i] \\
(\forall\tau_0)[\tau/i] & = \forall(\tau_0[\tau/i+1]) \\
j[\tau/i] & = \begin{cases} j-1 & \text{if } j > i; \\ U_0^i(\tau) & \text{if } j = i; \\ j & \text{if } j < i. \end{cases} \\
\langle\tau_0, ..., \tau_{n-1}\rangle[\tau/i] & = \langle\tau_0[\tau/i], ..., \tau_{n-1}[\tau/i]\rangle \\
\mathsf{int}[\tau/i] & = \mathsf{int}
\end{array}
$$

$$
\begin{array}{rl}
U_k^i(\tau_1 \to \tau_2) & = U_k^i(\tau_1) \to U_k^i(\tau_2) \\
U_k^i(\forall\tau) & = \forall(U_{k+1}^i(\tau)) \\
U_k^i(j) & = \begin{cases} j+i & \text{if } j > k; \\ j & \text{if } j \le k. \end{cases} \\
U_k^i(\langle\tau_0, ..., \tau_{n-1}\rangle) & = \langle U_k^i(\tau_0), ..., U_k^i(\tau_{n-1})\rangle \\
U_k^i(\mathsf{int}) & = \mathsf{int}
\end{array}
$$

Figure 3. Substitution over $\lambda_{\to}$ types in de Bruijn

(2001)). It employs an "update" function $U_k^i(\tau)$ whose effect is to adjust all indices greater than k (those are the free variables) by incrementing them by i. The form $\mathsf{shift}\ \Gamma$ denotes the context where $U_0^1(-)$ has been applied to every type in Γ.

3.2 Terms

Whereas type-level binders are represented with de Bruijn indices, all term-level binders are represented at first with higher-order abstract syntax, whether they abstract values (fix or let) or types (Λ).

```
data Exp t where
  Fix :: (Exp (s → t) → Exp s → Exp t) → Exp (s → t)
  App :: Exp (s → t) → Exp s → Exp t

  TpAbs :: (∀t. Exp (Subst s t Z)) → Exp (All s)
  TpApp :: Exp (All s) → Exp (Subst s t Z)
```

A Haskell term of type *Exp* t encodes a $\lambda_{\to}$ term satisfying a typing judgement $\Delta; \Gamma \vdash e : \tau$, where the context $\Delta; \Gamma$ is tracked implicitly by Haskell's type context.

A type abstraction $\Lambda\tau$ is represented as a polymorphic term that, when instantiated at a given type τ_2, assumes type $\tau_1[\tau_2/0]$. It is higher-order in the sense that the object-level type variable is represented by a Haskell type variable (bound by an implicit type abstraction.) The substitution itself is an application of a type function *Subst*, defined in the next section.

To illustrate, the $flip$ function:

$$flip = \Lambda\alpha.\ \Lambda\beta.\ \mathsf{fix}\ f\ (x : \langle\alpha, \beta\rangle).\ \langle x.1, x.0\rangle$$

is encoded as:

```
flip :: All (All ((Var (S Z), Var Z) → (Var Z, Var (S Z))))
flip = TpAbs (TpAbs (Fix (λf x →
         Pair (Snd x)(Fst x))))
```

3.3 Substitutions

The substitution and update functions encode directly as Haskell type families. As their definition involve arithmetic over indices, we also need to define type functions accordingly. The complete list of type functions, with their meaning, is as follows:

```
type family Subst t1 t2 i      — τ1[τ2/i]
type family U k i t            — U_k^i(τ)
type family Pred i             — i − 1
type family Add i j            — i + j
type family CMP i j t1 t2 t3   — { t1  if i < j;
                                 { t2  if i = j;
                                 { t3  if i > j.
```

$$
\begin{array}{ll}
(\textit{types}) & \tau ::= \forall\vec{\alpha}.\ \tau \to 0 \mid \alpha \mid \langle\tau_0, ..., \tau_{n-1}\rangle \mid \mathsf{int} \\
(\textit{values}) & v ::= \mathsf{fix}\ f[\vec{\alpha}]\ x.\ e \mid x \mid \langle e_0, ..., e_{n-1}\rangle \mid n \\
(\textit{exps}) & e ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \\
 & \quad \mid \mathsf{let}\ x = v.i\ \mathsf{in}\ e \mid v_1[\vec{\tau}]\ v_2 \mid \mathsf{if0}\ v\ e_1\ e_2 \\
 & \quad \mid \mathsf{halt}\ v
\end{array}
$$

Figure 4. Syntax of $\lambda_{\mathcal{K}}$

Object type	Haskell type	
$\forall\vec{\alpha}.\ \tau \to 0$	*Cont k t*	
$\exists\alpha.\tau$	*Exists t*	($\lambda_{\mathcal{C}}$ and $\lambda_{\mathcal{H}}$ only)
α	*Var i*	
$\langle\tau_0, ..., \tau_{n-1}\rangle$	$(t_0, (\ldots, (t_{n-1}, ()) \ldots))$	
int	*Int*	

Figure 5. Encoding of the types of $\lambda_{\mathcal{K}}$, $\lambda_{\mathcal{C}}$ and $\lambda_{\mathcal{H}}$

Henceforth, the definition of individual type families is straightforward:

```
type instance Subst All s t i = All (Subst s t (S i))
type instance Subst (Var j) t i =
    CMP i j (Var (Pred j)) (U Z i t) (Var j)
...

type instance U k i (All t) = All (U (S k) i t)
type instance U k i (Var j) = Var (CMP j k j j (Add j i))
...
```

4. CPS conversion

This section presents the salient features of our CPS conversion over the encoding of $\lambda_{\to}$ from the last section.

In particular, the amount of type annotations in this implementation is notably low, especially in the simply-typed fragment. It is essentially limited to annotating the type of the CPS conversion function, as the code itself requires no further annotation. Unfortunately, in its current state the treatment of polymorphism requires that we annotate the constructors of type abstraction and application with type representatives, in order to instantiate a lemma that captures the effect of the translation on type substitutions.

4.1 Target language

The syntax of the CPS language ($\lambda_{\mathcal{K}}$, shown in Fig. 4) is split into two categories of values and expressions. Accordingly we define two types:

```
data ValK t = ...

data ExpK = ...
```

The type *ValK* t encodes well-typed values of type τ (satisfying a judgment $\Delta; \Gamma \vdash_{\mathcal{K}} v : \tau$), and *ExpK* encodes well-typed expressions (satisfying a judgment $\Delta; \Gamma \vdash_{\mathcal{K}} e$). The representation is developed in the same manner as in the previous section, using a first-order encoding for types and a higher-order one for all term-level binders. For reference, the encoding of types is summarized in Fig. 5 (the type encoding is essentially the same for $\lambda_{\mathcal{K}}$ and the subsequent intermediate languages, with the exception of existential types, which are introduced by closure conversion.)

Whereas on paper it is simpler to have a single fix operator that abstracts type and term variables and provides recursion, it simplifies subsequent transformations somewhat to have a monomorphic fix operator and a separate type abstraction (which also abstracts a term variable, but is not recursive):

$$
\begin{array}{rcl}
\mathcal{K}_{\text{type}}[\![\tau_1 \to \tau_2]\!] & = & \langle \mathcal{K}_{\text{type}}[\![\tau_1]\!], \mathcal{K}_{\text{type}}[\![\tau_2]\!] \to 0\rangle \to 0 \\
\mathcal{K}_{\text{type}}[\![\forall\alpha.\ \tau]\!] & = & \forall\alpha.\ ((\mathcal{K}_{\text{type}}[\![\tau]\!]) \to 0) \to 0 \\
\mathcal{K}_{\text{type}}[\![\alpha]\!] & = & \alpha \\
\mathcal{K}_{\text{type}}[\![\langle\tau_1, ..., \tau_n\rangle]\!] & = & \langle \mathcal{K}_{\text{type}}[\![\tau_1]\!], ..., \mathcal{K}_{\text{type}}[\![\tau_n]\!]\rangle \\
\mathcal{K}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int} \\
\\
\mathcal{K}_{\text{prog}}[\![e]\!] & = & \mathcal{K}[\![e]\!]\ (\lambda x\ .\ \mathsf{halt}\ x) \\
\\
\mathcal{K}[\![x]\!]\ \kappa & = & \kappa\ x \\
\mathcal{K}[\![\mathsf{fix}\ f\ x.\ e]\!]\ \kappa & = & \kappa\ (\mathsf{fix}\ f\ \langle x, c\rangle.\ \mathcal{K}[\![e]\!]\ c) \\
\mathcal{K}[\![e_1\ e_2]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ x_1\ \langle x_2, \kappa\rangle)) \\
\mathcal{K}[\![\Lambda\alpha.\ e]\!]\ \kappa & = & \kappa\ (\lambda[\alpha]\ c.\ \mathcal{K}[\![e]\!]\ c) \\
\mathcal{K}[\![e[\tau]]\!]\ \kappa & = & \mathcal{K}[\![e]\!]\ (\lambda x\ .\ x[\mathcal{K}_{\text{type}}[\![\tau]\!]]\ (\lambda y\ .\ \kappa\ y)) \\
 & \ldots &
\end{array}
$$

Figure 6. CPS conversion

$$
\begin{array}{l}
\mathit{Kfix} :: (\mathit{ValK}\ (\mathit{Cont}\ Z\ s) \to \mathit{ValK}\ s \to \mathit{ExpK}) \\
\qquad\qquad \to \mathit{ValK}\ (\mathit{Cont}\ Z\ s) \\
\mathit{KtpAbs} :: (\forall t.\ \mathit{ValK}\ (\mathit{Subst}\ s\ t\ Z) \to \mathit{ExpK}) \\
\qquad\qquad \to \mathit{ValK}\ (\mathit{Cont}\ (S\ Z)\ s)
\end{array}
$$

where *Cont* k t is the Haskell type we use for $\forall\vec{\alpha}.\ \tau \to 0$, the parameter k reflecting the number of abstracted type variables.

4.2 Translation

The CPS conversion of types ($\mathcal{K}_{\text{type}}[\![-]\!]$), programs ($\mathcal{K}_{\text{prog}}[\![-]\!]$), and open terms ($\mathcal{K}[\![-]\!]\ -$) is shown in Fig.6. The type family that encodes $\mathcal{K}_{\text{type}}[\![-]\!]$ is defined as:

```
type family Ktype t
type instance Ktype (s → t) =
    Cont Z (Ktype s, Cont Z (Ktype t))
type instance Ktype (Var i) = Var i
type instance Ktype (All t) =
    Cont (S Z) (Cont Z (Ktype t))
...
```

Type preservation is reflected in the signature of the functions that implement $\mathcal{K}_{\text{prog}}[\![-]\!]$ and $\mathcal{K}[\![-]\!]\ -$:

```
cpsProg :: Exp t → ExpK
cps :: Exp t → (ValK (Ktype t) → ExpK) → ExpK
```

The type of *cpsProg* encodes the usual type preservation theorem, stating that the conversion takes well-typed $\lambda_{\to}$ programs (i.e. closed expressions) to well-typed $\lambda_{\mathcal{K}}$ programs:

THEOREM 4.1. *(CPS type preservation) If* $\bullet;\bullet \vdash e : \tau$ *then* $\bullet;\bullet \vdash_{\mathcal{K}} \mathcal{K}_{prog}[\![e]\!]$.

Similarly the type of *cps* embodies the theorem which states that $\mathcal{K}[\![-]\!]\ -$ takes well-typed expressions to well-typed expressions, provided that the supplied continuation has the expected type:[2]

LEMMA 4.1. *($\lambda_{\to}$ – $\lambda_{\mathcal{K}}$ type correspondence) If* $\Gamma \vdash e : \tau$ *and*

$$\Delta; \mathcal{K}_{type}[\![\Gamma]\!] \vdash_{\mathcal{K}} \lambda x\ .\ \kappa\ x : (\mathcal{K}_{type}[\![\tau]\!] \to 0) \to 0$$

then

$$\Delta; \mathcal{K}_{type}[\![\Gamma]\!] \vdash_{\mathcal{K}} \mathcal{K}[\![e]\!]\ \kappa.$$

Note that, since we use HOAS and the context $\Delta;\Gamma$ is implicit in our encoding, we get preservation of the type environment "for free". The situation will be opposite when we turn to closure conversion in Sec. 6.

[2] In this lemma the continuation κ lies at the meta level and must be wrapped into an object-level function so it can be the subject of a typing judgment.

4.3 Polymorphism

While the above stated theorems cover the theory of type preservation for simple types, polymorphism introduces issues of its own. The technical difficulty is to convince the type checker that we obtain a well-typed term when converting type applications (and abstractions), as it involves reconstructing a term whose type is defined by a substitution. The argument relies on the fact that our notion of substitution commutes with the conversion of types:

LEMMA 4.2. *($\mathcal{K}_{type}[\![-]\!]$–subst commute) For any $\lambda_{\to}$ types τ_1, τ_2 and index i,*

$$\mathcal{K}_{type}[\![\tau_1[\tau_2/i]]\!] = (\mathcal{K}_{type}[\![\tau_1]\!])[\mathcal{K}_{type}[\![\tau_2]\!]/i].$$

This means that we actually need to make coercion like:

```
    ValK (Ktype (Subst s t Z))
→   ValK (Subst (Ktype s) (Ktype t) Z)
```

Currently we do not have any way of doing such coercion purely at the type level, and we need to implement the lemma as a term-level functions that produces a witness that the coercion is valid (see (Guillemette and Monnier 2008) for alternative solutions to this problem). Its type is:

```
substCpsCommute ::
      TypeRep s → TypeRep t
  → Equiv (CPS (Subst s t Z)) (Subst (CPS s) (CPS t) Z)

data Equiv s t where
    Equiv :: s ∼ t ⇒ Equiv s t
```

The type *Equiv* uses another feature introduced in GHC along with type families, namely *type equality coercions* (Sulzmann et al. 2007). The context $(s \sim t)$ means that the types s and t, although possibly syntactically different, are equivalent after applying a process of normalization (which in particular eliminates applications of type functions.) The lemma itself (*substCpsCommute*) can be defined by case analysis over type representatives. Alternatively, it can do a dynamic test: it can construct a representation of the two types to prove equal and perform a comparison over them to supply evidence that they match.

Of course, in order to be able to apply the lemma in its current form, we need to annotate the syntax tree with type representatives. For instance, the data constructor for type application from Sec. 3.2 actually needs to bear representatives of the universal type and the type argument:

```
TpApp :: TypeRep s → TypeRep t
      → Exp (All s) → Exp (Subst s t Z)
```

Then, *cps* can call the lemma to get the required type assumption as needed:

```
cps (TpApp s t e) k =
  case substCpsCommute s t of
    Equiv → cps e (λx → KtpApp (kType s) (kType t) x
                               (lamK k))
```

where *kType* reifies *Ktype* at the term level:

```
kType :: TypeRep t → TypeRep (Ktype t)
```

Type abstraction. A consequence of the higher-order encoding of type abstraction is that the function $\mathcal{K}_{\text{type}}[\![-]\!]$ must be invertible. We need to convert the functional argument of *TpAbs* (say f) to that of *KtpAbs* (say f'). The function f' receives a representative of a type in CPS form, and constructs a representative of the originating

type in direct style so as to be able to apply f, and finally converts the resulting term back in CPS. To achieve this, we must define the inverse of $\mathcal{K}_{\text{type}}[\![-]\!]$ as a type family (and also reify this function at the term level to construct the representative):

```
type family UnKtype t
type instance UnKtype (Cont (S Z) (Cont Z t)) =
        All (UnKtype u)
...
unKtype :: TypeRep t → TypeRep (UnKtype t)
unKtype = ...
```

4.4 Implementation

HOAS. We actually use a concrete representation of $\lambda_{\to}$ and $\lambda_{\mathcal{K}}$ that differs from what we have shown so far, in a way that rules out exotic terms, and also makes the traversal of the syntax easier. We essentially use a type-annotated version of the HOAS encoding of Washburn and Weirich (2003). The absence of interaction between this aspect and type preservation allowed us to defer its discussion to this point.

To illustrate the difference, consider the constructor for let:

```
data Exp t where
  Let :: Exp t1 → (Exp t1 → Exp t2) → Exp t2
```

In the concrete representation, it takes the form:

```
data ExpF α t where
  Let :: α t1 → (α t1 → α t2) → α t2
type Exp α t = Rec ExpF α t
```

where *Rec* plays the role of a fixed-point type operator. A term of source type *t* is represented as a Haskell term of type $\forall\alpha.\ \mathit{Exp}\ \alpha\ t$ (where the parametricity in α rules out exotic terms.) The type *Exp* comes equipped with an elimination form (the "catamorphism"), whose type is:

```
cata :: (∀t. ExpF (β t) → β t)
     → (∀t. (∀α. Exp α t) → β t)
```

Intuitively, the type β stands for "the result of the computation" over the source term (indexed by source type). Here, we obtain *cps* by applying *cata* with $\beta\ t$ instantiated at the type:

```
type CPS α t = (ValK α (Ktype t) → ExpK α) → ExpK α
```

The function passed to *cata* visits a single node in the syntax tree, and has type:

```
cpsAux :: ExpF (CPS α t) → CPS α t
```

Danvy and Filinski's* CPS *transform. Our compiler actually implements the one-pass CPS conversion of Danvy and Filinski (1992), where administrative redexes are reduced on-the-fly. As shown by Washburn and Weirich (2003), it can be conveniently implemented by adding an extra component to the result of *cps*, that expects an object-level continuation (*cpsObj*) instead of a meta-level one (*cpsMeta*):

```
data CPS α t where
  CPS :: ((ValKα (Ktype t) → ExpK α) → ExpK α)
      → ((ValKα (Ktype t → Z)) → ExpK α)
      → CPS α t

cpsMeta e = case e of CPS meta _ → meta
cpsObj e = case e of CPS _ obj → obj
```

For example, the code that handles the conversion of fix is implemented as follows:

```
cpsAux (Fix f) =
  value (FixK (λself x →
    KletFst x (λarg →
      KletSnd x (λk →
        cpsObj (f (value self) (value arg)) k))))
```

where *value* is a function that CPS-converts a value:

```
value :: ValK α (Ktype t) → CPS α t
value v = CPS (λk → k v)          − cpsMeta
              (λc → Kapp c v)     − cpsObj
```

(*types*) $\tau ::= \forall\vec{\alpha}.\ \tau \to 0 \mid \exists\alpha.\ \tau \mid \alpha \mid \langle\tau_0, ..., \tau_{n-1}\rangle \mid \mathsf{int}$

(*values*) $v ::= \mathsf{fix}\ f[\vec{\alpha}]\ x.\ e \mid x \mid \mathsf{pack}\ (\tau_1, v)\ \mathsf{as}\ \exists\alpha.\ \tau_2 \mid v[\tau] \mid \langle e_0, ..., e_{n-1}\rangle \mid v.i \mid n$

(*exps*) $e ::= \mathsf{let}\ (\alpha, x) = \mathsf{unpack}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \mid v_1\ v_2 \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v$

Figure 7. Syntax of λ_C

5. Conversion to de Bruijn indices

Before closure conversion takes place, the $\lambda_{\mathcal{K}}$ program is converted to a first-order representation. The types *ValK* t and *ExpK* are mapped to types:

```
ValKb (i, ts) t
ExpKb (i, ts)
```

where i encodes Δ (i.e. it reflects the number of type variables in scope) and *ts* encodes Γ (i.e. it lists the types of the term variables in scope.) Also for instance the data constructors *Kfix* and *KtpAbs* from Sec. 4.1 are translated to these two:

```
KBfix :: ExpKb (i, (s, (Cont Z s, ts)))
      → ValKb (i, ts)(Cont Z s)
KBtpAbs :: ExpKb (S i, (s, Shift ts))
        → ValKb (i, ts) (Cont (S Z) s)
```

The type of *KBtpAbs* makes explicit that the body of the type abstraction has an extra type variable in scope (and that its term context is adjusted accordingly.) Note that term-level type variables are not anymore encoded in HOAS.

5.1 Translation

The conversion to de Bruijn indices constructs a representation whose type reflects the term's context $(\Delta; \Gamma)$ and replaces all variables occurrences with indices. The conversion passes around a representation of the context, which gets constructed as the conversion proceeds:

```
toBv :: ValK t → NatRep i → EnvRep ts → ValKb (i, ts) t
toBe :: ExpK → NatRep i → EnvRep ts → ExpKb (i, ts)
```

When converting a variable occurrence, the term variable context where the variable occurs (Γ) is compared to the context where it was bound (Γ_0); the difference in length between these two indicates which index to substitute for the variable. As the type of a variable is not related to the type variable context (Δ), the introduction of polymorphism did not influence this phase significantly. The original implementation (restricted to the simply typed case) is discussed in more detail in (Guillemette and Monnier 2007).

6. Closure conversion

The target language of closure conversion (λ_C, shown in Fig. 7) forces fix values to be closed and introduces existential types; it also

moves type applications to the syntactic class of values, and allows tuple projections to appear as values as well, which simplifies the conversion slightly.

De Bruijn indices are used for all binders in this representation, defined by the two types:

data*ValC* (i, ts) t

data *ExpC* (i, ts)

The fix operator of λ_C binds a number of type variables (in addition to the function's argument and the binder for the recursive call.) The typing rule for fix forces the function to be closed, i.e. exempt of free term or type variables:

$$\frac{\alpha_1, ..., \alpha_n; x : \tau, f : \forall\alpha_1, ..., \alpha_n.\ \tau \to 0 \vdash_C e}{\Delta; \Gamma \vdash_C \mathsf{fix}\ f[\alpha_1, ..., \alpha_n]\ x.\ e : \forall\alpha_1, ..., \alpha_n.\ \tau \to 0}$$

This time the actual representation includes a single constructor that directly encodes the typing rule for fix:

$$\begin{array}{rl} \mathit{Cfix} :: & \mathit{ExpC}\ (k, (t, (\mathit{Cont}\ k\ t, ()))) \\ \to & \mathit{ValC}\ (i, ts)\ (\mathit{Cont}\ k\ t) \end{array}$$

This type reflects the closedness conditions: the body's term variable context finishes with $()$, meaning that it cannot have free term variables, and since t appears in a context where k variables are in scope, t cannot involve type variables other than those bound by the fix.

Existentials. The language introduces existential types, which are used to abstract the type of the environment when forming closures.

$$\begin{array}{rl} \mathit{Cpack} :: & \mathit{ValC}\ (i, ts)\ (\mathit{Subst}\ s\ t\ Z) \\ \to & \mathit{ValC}\ (i, ts)\ (\mathit{Exists}\ s) \\ \mathit{Cunpack} :: & \mathit{ValC}\ (i, ts)\ (\mathit{Exists}\ s) \\ \to & \mathit{ExpC}\ (S\ i, (s, \mathit{Shift}\ ts)) \\ \to & \mathit{ExpC}\ (i, ts) \end{array}$$

In much the same way that we did for universal types, the type of these constructors encode the usual typing rules for existential types:

$$\frac{\Delta; \Gamma \vdash_C v : \tau_2[\tau_1/\alpha]}{\Delta; \Gamma \vdash_C \mathsf{pack}\ (\tau_1, v)\ \mathsf{as}\ \exists\alpha.\ \tau_2 : \exists\alpha.\ \tau_2}$$

$$\frac{\Delta; \Gamma \vdash_C v : \exists\alpha.\ \tau \quad \alpha, \Delta; x : \tau, \Gamma \vdash_C e}{\Delta; \Gamma \vdash_C \mathsf{let}\ (\alpha, x)\ =\ \mathsf{unpack}\ v\ \mathsf{in}\ e}$$

6.1 Translation

The closure conversion of types ($\mathcal{C}_{\text{type}}[\![-]\!]$), values ($\mathcal{C}_{\text{val}}[\![-]\!]m$), and expressions ($\mathcal{C}_{\text{exp}}[\![-]\!]m$) is shown in Fig. 8.

The type translation for continuations introduces an existential variable β that abstracts the type of the closure environment and pairs up the function (which is made to receive the environment as an extra argument) with the environment.

type family *Ctype* t
type instance *Ctype* $(\mathit{Cont}\ i\ t) =$
$\quad \mathit{Exists}\ (\mathit{Cont}\ i\ (U\ (S\ Z)\ i\ (\mathit{Ctype}\ t), \mathit{Var}\ i), \mathit{Var}\ Z)$

Note that the update function must be applied in order to prevent free type variables from being captured by the existential quantifier.

The translation of values and expressions receives an extra parameter m, which maps every variable in scope to the expression used to access it[3]. In general, this expression is either a local variable or a projection of the environment.

[3] This map is necessary as a consequence of using de Bruijn indices; a formulation with variable names can manage without it.

$$\begin{array}{rcl} \mathcal{C}_{\text{type}}[\![\forall\vec{\alpha}.\ \tau \to 0]\!] & = & \exists\beta.\ \langle\forall\vec{\alpha}.\ \langle\mathcal{C}_{\text{type}}[\![\tau]\!], \beta\rangle \to 0, \beta\rangle \\ \mathcal{C}_{\text{type}}[\![\alpha]\!] & = & \alpha \\ \mathcal{C}_{\text{type}}[\![\langle\tau_0, ..., \tau_{n-1}\rangle]\!] & = & \langle\mathcal{C}_{\text{type}}[\![\tau_0]\!], ..., \mathcal{C}_{\text{type}}[\![\tau_{n-1}]\!]\rangle \\ \mathcal{C}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int} \end{array}$$

$$\begin{array}{rcl} \mathcal{C}_{\text{val}}[\![x]\!]m & = & \mathit{lookup}\ m\ x \\ \mathcal{C}_{\text{val}}[\![(\mathsf{fix}\ f[\vec{\alpha}]\ x.\ e)^{\tau}]\!]m & = & \mathsf{pack}\ (\tau_{env}, \langle v_{code}[\vec{\beta}], v_{env}\rangle) \\ & & \mathsf{as}\ \mathcal{C}_{\text{type}}[\![\tau]\!] \end{array}$$

$$\begin{array}{l} \text{where } \vec{\beta} = \mathit{ftvs}\ (\mathsf{fix}\ f[\vec{\alpha}]\ x.\ e) \\ \quad y_0^{\tau_0}, ..., y_{n-1}^{\tau_{n-1}} = \mathit{fvs}\ (\mathsf{fix}\ f[\vec{\alpha}]\ x.\ e) \\ \quad \tau_{env} = \langle\mathcal{C}_{\text{type}}[\![\tau_0]\!], ..., \mathcal{C}_{\text{type}}[\![\tau_{n-1}]\!]\rangle \\ \quad v_{code} = \mathsf{fix}\ f[\vec{\beta}, \vec{\alpha}]\ x. \\ \qquad \mathsf{let}\ x' = x.0 \\ \qquad\quad env = x.1 \\ \qquad\quad f' = \mathsf{pack}\ (\tau_{env}, \langle f, env\rangle)\ \mathsf{as}\ \mathcal{C}_{\text{type}}[\![\tau]\!] \\ \qquad \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!](x \Rightarrow x', f \Rightarrow f', \\ \qquad\qquad y_0 \Rightarrow env.0, ..., \\ \qquad\qquad y_{n-1} \Rightarrow env.(n-1)) \\ \quad v_{env} = \langle \mathit{lookup}\ m\ y_0, ..., \mathit{lookup}\ m\ y_{n-1}\rangle \end{array}$$

$$\begin{array}{rcl} \mathcal{C}_{\text{exp}}[\![v_1[\tau_1, ..., \tau_n]\ v_2]\!]m & = & \mathsf{let}\ (\alpha, x) = \mathsf{unpack}\ \mathcal{C}_{\text{val}}[\![v_1]\!]m \\ & & \quad x_f = x.0 \\ & & \quad x_{env} = x.1 \\ & & \mathsf{in}\ x_f[\mathcal{C}_{\text{type}}[\![\tau_1]\!], ..., \mathcal{C}_{\text{type}}[\![\tau_n]\!]] \\ & & \quad \langle\mathcal{C}_{\text{val}}[\![v_2]\!]m, x_{env}\rangle \end{array}$$

Figure 8. Closure conversion

$$\begin{array}{rl} ccV :: & \mathit{ValKb}\ (i, ts)\ t \\ \to & (\forall ts'.\ \mathit{MapT}\ (\mathit{Cenv}\ ts)\ (\mathit{ValC}\ (i, ts'))) \\ \to & \mathit{ValC}\ (i, ts')\ (\mathit{Ctype}\ t)) \\ ccE :: & \mathit{ExpKb}\ (i, ts) \\ \to & (\forall ts'.\ \mathit{MapT}\ (\mathit{Cenv}\ ts)\ (\mathit{ExpC}\ (i, ts'))) \\ \to & \mathit{ExpC}\ (i, ts') \end{array}$$

Informally, these types mean that the conversion takes a $\lambda_{\mathcal{K}}$ value (or expression) in context $\Delta; \Gamma$, to a $\lambda_{\mathcal{C}}$ value of the converted type (or an expression) in any context $\Delta; \Gamma'$, provided that the supplied map takes every term variable in Γ to a value of the converted type in $\Delta; \Gamma'$. Formally:

LEMMA 6.1. *(CC type correspondence) If* $\Delta; \Gamma \vdash_{\mathcal{K}} v : \tau$ *and*

$$\forall x \in \mathit{dom}(\Gamma).\ \Delta; \Gamma' \vdash_C \mathit{lookup}\ m\ x : \mathcal{C}_{\mathit{type}}[\![\Gamma\ x]\!]$$

then

$$\Delta; \Gamma' \vdash_C \mathcal{C}_{\mathit{val}}[\![v]\!]m : \mathcal{C}_{\mathit{type}}[\![\tau]\!].$$

The details of how the map is represented and how it is constructed when closures are formed (in the simply typed case) are spelled out in (Guillemette and Monnier 2007), and are not repeated here, as they are not much affected by polymorphism.

6.2 Polymorphism

Forming closures. By the definition of $\mathcal{C}_{\text{val}}[\![-]\!]m$, the function stored inside a closure is closed w.r.t type variables: it is made to take an extra set of type variables $\vec{\beta}$ that are the original function's free type variables. When forming the closure, the closed function is passed the free type variables, so as to get a closure of the expected type. The way this "forwarding" of type variables preserves types is captured by this simple lemma:

LEMMA 6.2. *(forwarding) If* $\vec{\beta} \subseteq \Delta$ *and*

$$\Delta; \Gamma \vdash_C v : \forall\vec{\beta}, \vec{\alpha}.\ \tau \to 0$$

then

$$\Delta; \Gamma \vdash_C v[\vec{\beta}] : \forall \vec{\alpha}.\, \tau \to 0.$$

In our implementation, *all* type variables in scope are captured when forming a closure, rather that just those that actually appear free in the function (that is, we take $\vec{\beta} = \Delta$.) It would require additional data structures and type families to perform free type variable analysis, and afterward selectively abstract and apply those variables (and it's far from obvious that it could be done in a convincing way.) In contrast, capturing all the type variables can be done directly. Then, their application ($v[\vec{\beta}]$) is constructed by a function that implements the forwarding lemma:

$$\begin{aligned} \mathit{tpAppMulti} :: \quad & \mathit{ValC}\ (j, ts)\ (\mathit{Cont}\ (\mathit{Add}\ i\ k)\ t) \\ \to\ & \mathit{ValC}\ (j, ts)\ (\mathit{Cont}\ k\ t) \end{aligned}$$

It is a simple recursive function that applies the topmost index as many times as needed, using the obvious fact that $\tau[0/0] = \tau$.

Type application. When translating the application of a polymorphic function, it takes a few manipulations to show that the function is of a type compatible with its supplied argument. Expectedly, we need again that substitution commute with the type translation:

LEMMA 6.3. *($\mathcal{C}_{type}[\![-]\!]$–subst commute) For any λ_K types τ_1, τ_2 and index i,*

$$\mathcal{C}_{type}[\![\tau_1[\tau_2/i]]\!] = (\mathcal{C}_{type}[\![\tau_1]\!])[\mathcal{C}_{type}[\![\tau_2]\!]/i].$$

As the type translation explicitly shifts indices, we also need similar lemmas showing that $\mathcal{C}_{\text{type}}[\![-]\!]$, $U^i_k(-)$, and substitution commute pairwise.

7. Hoisting

The hoisting phase moves functions, which are closed as a result of closure conversion, to a top-level letrec construct. It proceeds by collecting every function into a bundle (whose type reflects the type of every function in it) and then assembles the program. As the *collect* phase encounters an individual function, an index (to be bound by the letrec) is substituted in place of it, and the bundle of collected functions is extended with the new one (in which functions have already been collected and replaced with indices.) The indices that are previously assigned are "weakened" so as to make sense in the context of the extended bundle.

The target language of hoisting has an separate context (Γ', the type parameter *fs*) for the variables bound by the letrec, in addition to the usual one ($\Delta; \Gamma$):

$$\mathit{ValH}\ (i, ts, fs)\ t$$
$$\mathit{ExpH}\ (i, ts, fs)$$

The introduction of polymorphism has the consequence that the type of individual functions in the bundle reflect the number of universal quantifier in each function's type. Adding recursion actually had a stronger impact: as a function may refer to itself, we have to arrange for such references to be turned into the corresponding index bound by the letrec. For this purpose we use a map argument (analogous to the one used for closure conversion.) It maps each term variable in scope to a variable in the target program, which is bound by the letrec in the case of recursive occurrence and locally bound for other variables. The functions that collects fix expressions has type:

$$\begin{aligned} \mathit{collectV} :: \quad & \mathit{ValC}\ (i, ts)\ t \\ \to\ & \mathit{MapT}\ ts\ (\mathit{ValH}\ (i, ts, hs_0)) \\ \to\ & \exists hs.\ (\mathit{ValH}\ (i, ts, \mathit{Cat}\ hs_0\ hs)\ t, \\ & \quad \mathit{TupleH}\ (\mathit{Cat}\ hs_0\ hs)\ hs) \end{aligned}$$

$$\begin{aligned} \mathit{collectE} :: \quad & \mathit{ExpC}\ (i, ts) \\ \to\ & \mathit{MapT}\ ts\ (\mathit{ValH}\ (i, ts, hs_0)) \\ \to\ & \exists hs.\ (\mathit{ExpH}\ (i, ts, \mathit{Cat}\ hs_0\ hs), \\ & \quad \mathit{TupleH}\ (\mathit{Cat}\ hs_0\ hs)\ hs) \end{aligned}$$

The type variable hs_0 reflects the type of the bundle prior to the call, and *hs* that of the segment of the bundle to be appended as a result of visiting the current term. Here, *Cat* is a type family to handle concatenation of lists of types:

$$\begin{aligned} &\textbf{type family}\ \mathit{Cat}\ ts_0\ ts \\ &\textbf{type instance}\ \mathit{Cat}\ ()\ ts' = ts' \\ &\textbf{type instance}\ \mathit{Cat}\ (s, ts)\ ts' = (s, \mathit{Cat}\ ts\ ts') \end{aligned}$$

which must be accompanied with an associativity lemma.

Compared to CPS or closure conversion, the hoisting phase is by far the most conceptually simple, yet its implementation is the least succint of the three, due to its existentially quantified return type.

8. Experience and Future work

Identifying the right program representations is perhaps the most delicate part of this work. In this section we make some remarks concerning the choices we have made, and some alternatives that we did consider which turned out to be inadequate. Of course, identifying the right types to use is also of deep consequence, so we comment on our use of GADTs and type families, as well as other features that we used or considered using in past versions of the compiler.

8.1 Representing variables

When working on System F we considered many options for representing variables, both at the level of types ($\forall$, $\exists$, etc.) and terms (λ, let, Λ, unpack, etc.)

Type-level type variables. For type-level type variables, given the fact that we do not need to analyze those types, the best choice would assuredly by HOAS. But since GHC does not support type-level λ expressions, this is not an option.

From our experience, a de Bruijn encoding of types combined with type families for type-level operations (such as substitution) provides a fairly reasonable representation. It is also a fairly common choice in compilers using a typed intermediate language.

Using names would not be a good solution because it would be at least as complex as de Bruijn, with the added problem that α-equivalence is needed.

Term variables. For term variables, we started using HOAS, which is rather uncommon and is poorly supported in most languages, but served us well for the CPS transform. It is arguably more elegant than de Bruijn indices, and requires fewer type annotations as the typing environment is treated implicitly. When put in context though, it is hard to justify the choice of HOAS from an engineering standpoint, as it forces us to convert to and from first-order representations.

For the closure conversion and hoisting, on the other hand, HOAS cannot be used because of its inability to express that a term is closed.

The more common representation of term variables in compilers is as names, usually represented as small integers or as pointers, so that would be our favorite choice, but reifying small integers as singleton types to reason about them is rarely supported and GHC is no exception. Even using less efficient representations of names,

which lend themselves to singleton types, still suffers from the extra complexity of having to reason about freshness.

So we ended up using de Bruijn indices. As demonstrated, they do work, but they require delicate index updates at various places and accompanying lemmas, for example when moving code into or out of a scope, which phases like closure conversion and hoisting do all the time. The complexity we have in our current code is bearable, but we had to fine-tune it to get there: e.g., some apparently minor changes to the definition of *KBtpAbs* or *Cfix* can lead to a very significant increase of complexity. Another problem with de Bruijn indices is that most people find them mind-boggling to debug, although this is more true in untyped settings.

The representation of term-level variables for a compiler like ours is still a problem in search of a satisfactory solution. The most promising development on the horizon is probably (Pientka 2008; Pientka and Dunfield 2008).

Term-level type variables. If the term encoding is in HOAS, then the best option for type variables is to use HOAS as well, so that is what we have done. Using de Bruijn for type variables would not work: in de Bruijn the representation of a given (open) type depends on where that type appears in the term (i.e. how many Λ's have been traversed) – doing type instantiation by merely replicating the type using an application in the host language will not account for that.

If the term encoding is first-order, then HOAS may be a very good choice, (if the host language's monomorphism restriction does not get in the way), but in our case, for the same reason we could not use HOAS for term variables, we could not use HOAS for term-level type variables during closure conversion: we need to express the fact that the functions we output are also closed with respect to types.

Compilers tend to avoid de Bruijn indices in favor of names for term-level type variables, again in order to avoid the issues linked with shifting indices when moving code into or out of a scope. But we again decided to use de Bruijn indices for the same reason as for term variables: names are difficult to reify efficiently as types in GHC, and reasoning about freshness would introduce a lot of complexity and force us to restructure the code significantly.

In other words, essentially the same arguments that led us to choose de Bruijn indices for the term variables, led us to use de Bruijn for term-level type variables. And again, although we believe this choice to be the best there is, it is not satisfactory.

8.2 Type families

Before type families were made available in GHC, we used GADTs to encode witnesses of type preservation (Guillemette and Monnier 2006, 2007). Essentially, every time a term was produced, it was accompanied by a witness that the term was of the expected type. The drawbacks of this scheme are run-time overhead, a substantial amount of code bloat (for manipulating the existential packages), and the fact that our "proofs" were encoded in an unsound logic. Type families essentially solved these problems. We further compare the schemes that use only GADTs or GADTs plus type families in (Guillemette and Monnier 2008).

The representation of System F also benefited from type families. In the past we actually worked on a representation which relied on GADTs to encode witnesses of type applications (essentially encoding the type families from Sec. 3 as GADTs). Type families obviously make this representation much more direct.

8.3 Lemmas over type families

As seen in Sec. 4.3, we need to prove properties of the type families we define for the CPS and closure conversion of System F to type-check. Our current implementation implements such lemmas as term-level functions that produce a proof witness that the lemma holds at particular types. This is unsatisfactory in a number of ways: it incurs run-time overhead, it forces us to carry type-representatives as part of the syntax trees, and it encodes the lemma in an unsound logic (due to non-termination at the term level).

An alternative approach is to annotate the syntax tree with equality constraints that state the lemma holds at the given types.

$$\mathit{TpApp} :: \mathit{CPS}\ (\mathit{Subst}\ s\ t\ Z) \sim \mathit{Subst}\ (\mathit{CPS}\ s)\ (\mathit{CPS}\ t)\ Z \Rightarrow \mathit{Exp}\ (\mathit{All}\ s) \rightarrow \mathit{Exp}\ (\mathit{Subst}\ s\ t\ Z)$$

This is a purely static solution – it does not suffer from the unsoundness issue, and has no run-time overhead. The drawback is that, as such constraints annotate the syntax tree of the source language, it must be constructed in the compiler front-end and propagated to the point where it is use, thereby limiting modularity in the compiler.

To address this limitation, we are investigating a language extension to directly support lemmas over type families: to have the type-checker verify that all known instances satisfy the lemmas declared by the programmer (Guillemette and Monnier 2008). This way we could get the type equality coercion we need without having to encode it explicitly in the syntax tree.

8.4 Type Classes

Having started this work from an existing untyped compiler using algebraic data types for its term representation, it was only natural to use GADTs. This said, there is no indication that the same could not be done with multi-parameter type classes, but GADTs are probably a more natural representation for abstract syntax trees in a functional language.

Early on, we tried to use type classes to encode type-level functions as well as various proof objects. This was meant to help us by letting the type checker infer more of the type annotations and hence leave us with a cleaner code more focused on the actual algorithm than on the type preservation proof. Sadly we bumped into serious difficulties due to the fact that the then current version of GHC was not yet able to properly handle tight interactions been GADTs and type classes. More specifically the internal language of GHC had limitations that prevented some "exotic" uses of functional dependencies. Those limitations can appear without GADTs, but in our use of GADTs, we bumped into them all the time. In the mean time, type families appeared and provided an alternative way to let the type system and type inference do more of the work.

The shift to F_C (Sulzmann et al. 2007) as the internal language in GHC potentially improves the interaction between GADTs and type classes. Yet, as we discussed elsewhere (Guillemette and Monnier 2008), using type classes to prove type preservation necessitates extra annotations (in the form of class constraints) on the syntax tree, which must be propagated from phase to phase and would compromise modularity.

8.5 Future work

Our compiler is still in development. It lacks register allocation, instruction selection, and optimization phases. Also the source language still needs to be expanded with recursive types and existential types, as well as some encoding of algebraic data types. The intention being to accept as input a language comparable to GHC's internal System F-like language, so as to be able to bootstrap.

Regarding the part of the compiler already implemented, we hope to find some clean way to move the unsound term-level proofs (such as the implementation of our commutativity lemmas as functions) to the sound (and cheaper) type-level.

In the longer run, we may want to investigate how to generate PCC-style proofs. Since the types are not really propagated any more during compilation, constructing a PCC-style proof would probably need to use a technique reminiscent of (Hamid et al. 2002): build them separately by combining the source-level proof

of type-correctness with the verified proof of type preservation somehow extracted from the compiler's source code.

9. Related work

There has been a lot of work on typed intermediate languages, beginning with TIL (Tarditi et al. 1996) and FLINT (Shao and Appel 1995; Shao 1997), originally motivated by the optimizations opportunities offered by the extra type information. The idea of Proof-Carrying Code (Necula 1997) made it desirable to propagate type information even further than the early optimization stages, as is done in the setting of typed assembly language (Morrisett et al. 1999).

Shao et al. (2002) show a low-level typed intermediate language for use in the later stages of a compiler, and more importantly for us, they show how to write a CPS translation whose type-preservation property is statically and mechanically verified, like ours.

Pašalić (2004) constructed a statically verified type-safe interpreter with staging for a language with binding structures that include pattern matching. The representation he uses is based on de Bruijn indices and relies on type equality proofs in Haskell.

Chiyan Chen et al. (2003) also show a CPS transformation where the type preservation property is encoded in the meta language's type system. They use GADTs in similar ways, including to explicitly manipulate proofs, but they have made other design tradeoffs: their term representation is first order using de Bruijn indices, and their implementation language is more experimental. In a similar vein, Linger and Sheard (2004) show a CPS transform over a GADT-based representation with de Bruijn indices; but in contrast to Chen's work and ours, they avoid explicit manipulation of proof terms by expressing type preservation using type-level functions. We showed the CPS phase of our compiler in an earlier article (Guillemette and Monnier 2006), where the distinguishing feature is the use of a term representation based on HOAS.

Chlipala's compiler (2007) developed in the Coq proof assistant and ran as an extracted OCaml program, has a completely formalized correctness proof. Like ours, it compiles a higher-order, simply typed functional language (with similar code transformations) and uses de Bruijn representations throughout all phases. He uses a language whose type system is much more powerful than ours, but whose computational language is more restrictive.

Similarly, Leroy's compiler (Leroy 2006) for a first-order (C-like) language, written in the Coq proof assistant, has a completely formalized correctness proof.

A detailed proof of type preservation for an earlier formulation of closure conversion over System F was formulated by Minamide et al. (1996). Defunctionalization over a superset of System F in shown to be type-preserving by Pottier and Gauthier (2004).

Fegaras and Sheard (1996) showed how to handle higher-order abstract syntax, and Washburn and Weirich (2003) showed how to use this technique in a language such as Haskell. We use this latter technique and extend it to GADTs and to monadic catamorphisms.

References

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, aug 2006.

Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 20–28, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-667-6.

James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Symposium on Programming Languages Design and Implementation*, pages 54–65. ACM Press, June 2007.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*. ACM Press, September 2008.

Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2 (4):361–391, 1992.

Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in Haskell. In *Programming Languages meets Program Verification*, volume 174(7) of *Electronic Notes in Theoretical Computer Science*, pages 23–39, August 2006.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in Haskell. In *Haskell Workshop*. ACM Press, September 2007.

Louis-Julien Guillemette and Stefan Monnier. One vote for type families in Haskell! In *The 9th symposium on Trends in Functional Programming*, 2008.

Nadeem Abdul Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

Fairouz Kamareddine. Reviewing the classical and the de bruijn notation for λ-calculus and pure type systems. *Journal of Logic and Computation*, 11, 2001.

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages*, pages 42–54, New York, NY, USA, January 2006. ACM Press. ISBN 1-59593-027-2.

Nathan Linger and Tim Sheard. Programming with static invariants in Omega. Unpublished, 2004.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–283, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

Emir Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health and Sciences University, The OGI School of Science and Engineering, 2004.

Simon Peyton-Jones et al. The Haskell Prime Report. Working Draft, 2007.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Symposium on Principles of Programming Languages*, pages 371–382, 2008.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *Symposium on Principles and Practice of Declarative Programming*, 2008.

François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. *SIGPLAN Not.*, 39(1):89–98, 2004. ISSN 0362-1340.

Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel M. T. Chakravarty. Towards open type functions for Haskell. Presented at IFL 2007, 2007.

Zhong Shao. An overview of the FLINT/ML compiler. In *International Workshop on Types in Compilation*, June 1997.

Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Symposium on Programming Languages Design and Implementation*, pages 116–129, La Jolla, CA, June 1995. ACM Press.

Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, January 2002.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-605-6.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation*, January 2007.

David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Symposium on Programming Languages Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996. ACM Press.

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 249–262, Uppsala, Sweden, August 2003. ACM SIGPLAN.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, January 2003.