Statically Verified Type-Preserving Code Transformations in Haskell

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

Abstract

The use of typed intermediate languages can significantly increase the reliability of a compiler. By type-checking the code produced at each transformation stage, one can identify bugs in the compiler that would otherwise be much harder to find. We propose to take the use of types in compilation a step further by verifying that the transformation itself is type correct, in the sense that it is impossible that it produces an ill typed term given a well typed term as input.

We base our approach on higher-order abstract syntax (HOAS), a representation of programs where variables in the object language are represented by meta-variables. We use a representation that accounts for the object language's type system using generalized algebraic data types (GADTs). In this way, the full binding and type structure of the object language is exposed to the host language's type system. In this setting we encode a type preservation property of a CPS conversion in Haskell's type system, using witnesses of a type correctness proof encoded as values in a GADT.

1 Introduction

While there is still a long way to go until they become as common place as in digital systems, formal methods are rapidly improving and gaining ground in software. Type systems are arguably the most successful and popular formal method used to develop software, even more so since the rise of Java. For this reason, there is a lot of interest in trying to beef up type systems incrementally to enable them to prove more complex properties.

Thus 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. After all, why bother proving any property of our source code, if our compiler can turn it into some unrelated machine code? One of the opportunities is to use types to address this need. This is what we are trying to do.

Typed intermediate languages have been used in compilers for various purposes such as type-directed optimization [8, 23, 15], sanity checks to help catch compiler errors, and more recently to help construct proofs that the generated code verifies some properties [11, 6]. Typically the source level types are represented in those typed representations in the form of data-structures which have to be carefully manipulated to keep them in sync with the code they annotate as this code progresses through the various stages of compilation. This has several drawbacks:

- Additional work, obviously, which can slow down our compiler. To minimize the impact, the type language and the type annotations have to be very carefully designed and coded, using techniques like hash-consing, explicit substitutions, and other optimizations [18].
- Occasionally, the need to update the type annotations can make an optimization impractical, e.g. because the necessary type information is not immediately available and thus requires restructuring the algorithm.
- Need to choose between different design tradeoffs: either place only as few type annotations as possible to reduce the impact of the first problem above, or on the contrary, add type annotations everywhere to reduce the risk of bumping into the second problem above.
- Errors are only detected when we run the type checker, but running it as often as possible slows down our compiler even more.
- This amounts to *testing* our compiler, thus bugs can lurk, undetected.

To avoid those problems, we want to represent the source types of our typed intermediate language as types instead of data. This way the type checker of the language in which we write our compiler can *verify* once and for all that our compiler preserves the typing correctly. The compiler itself can then run at full speed without having to manipulate and check any more types. Also this gives us even earlier detection of errors introduced by an incorrect program transformation, and at a very fine grain, since it amounts to running the type checker after every instruction rather than only between phases.

The type-preservation argument has been introduced into the implementation of a compiler using a typeful program representation in [2]. But to our knowledge, the work presented here is the first attempt to formally establish a type preservation property using a language so widely used and well supported as Haskell, for which a industrial strenth compiler is available.

This work follows a similar goal to the one of [9], but we only try to prove the correctness of our compiler w.r.t the static semantics rather than the full dynamic semantics. In return we want to use a more practical programming language and hope to limit our annotations to a minimum such that the bulk of the code should deal with the compilation rather than its proof. Also we have started this work from the frontend and are making our way towards the backend, whereas Leroy's work has started with the backend. Our contributions are the following:

- We show a type-preserving CPS translation written in Haskell and where the GHC compiler verifies the property of type-preservation.
- We extend the classical toy example of a generalized algebraic data type (GADT) representation of an abstract syntax tree, to a full language with bindings.
- We use higher-order abstract syntax (HOAS) in our intermediate representation, following [24], and we show how to combine this technique with GADTs and how to build such terms using Template Haskell.

The remainder of this paper is organized as follows. We review generalized algebraic datatypes and the notion of higher-order abstract syntax in Sec. 2. Section 3 presents the CPS conversion, states a type-preservation property that it satisfies, and then shows how we encoded it in Haskell. Section 4 presents some alternative approaches, as well as some solutions to some of the problems we encountered. Section 5 mentions related work and Sec. 6 concludes.

2 Background

In this section we develop a typeful program representation using GADTs and higher-order abstract syntax for a simple source language that is a simply-typed λ-calculus with pairs and integers (herein called $\lambda_{\rightarrow}$.) We briefly describe the programming techniques used for manipulating such a representation based on Washburn and Weirich's work [24].

2.1 Generalized algebraic datatypes

Generalized algebraic datatypes (GADTs) [25, 3] 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 GADTs to represent abstract syntax trees, and use these type annotations to track the source-level type of an expression.

Consider the language $\lambda_{\rightarrow}$ defined in Fig. 1. The fragment of $\lambda_{\rightarrow}$ concerned with integers could be represented in a GADT as follows:

```
data Exp t where
  Num   :: Int                          -> Exp Int
  Prim  :: PrimOp -> Exp Int -> Exp Int -> Exp Int
  If0   :: Exp Int -> Exp t -> Exp t    -> Exp t

data PrimOp = Add | Sub | Mult
```

This `Exp` data type not only defines the abstract syntax but also encodes the typing rules of our language. E.g. a statement such as $\Gamma \vdash e : \tau$ is represented in Haskell by the fact that `e` :: `Exp` τ. The environment Γ is kept implicit.

$$
\begin{array}{ll}
(\textit{types}) & \tau ::= \tau_1 \to \tau_2 \mid \mathsf{int} \mid \tau_1 \times \tau_2 \\
(\textit{type env}) & \Gamma ::= \bullet \mid \Gamma, x{:}\tau \\
(\textit{primops}) & p ::= + \mid - \mid \times \\
(\textit{exps}) & e ::= x \mid \lambda x{:}\tau_1.\, e : \tau_2 \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_i\ e \mid i \mid e_1\ p\ e_2 \\
 & \quad \mid \mathsf{if0}\ e_1\ e_2\ e_3
\end{array}
$$

Typing rules

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_2 : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i} \qquad \frac{}{\Gamma \vdash i : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1\ p\ e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if0}\ e_1\ e_2\ e_3 : \tau}$$

Figure 1: $\lambda_{\to}$ syntax and static semantics

Note also that an expression of type `Exp t` represents a $\lambda_{\to}$ expression of *source type* t, where we have (arbitrarily) chosen the Haskell type `t` to stand for the corresponding $\lambda_{\to}$ type t (e.g. we use the Haskell type `Int` to represent the $\lambda_{\to}$ type int.)

Extending this encoding for the variable and the λ cases is not straightforward since we kept Γ implicit. Of course, we could try to make Γ explicit as in `Exp t` Γ, but that can quickly become cumbersome since it can entail reifying variables at the level of types, and encoding structural rules such as weakening and exchange. So instead, we use higher-order abstract syntax which allows us to keep Γ implicit.

2.2 Higher-order abstract syntax

Higher-order abstract syntax (HOAS) [14] is a program representation where variables in the object language are represented using meta-variables. For instance, functions in our source language would be represented using Haskell functions; thus we could extend the representation of the language to account for $\lambda_{\to}$ functions as follows:

```
data Exp t where
  ...
  Lam   :: (Exp s -> Exp t)             -> Exp (s -> t)
  App   :: Exp (s -> t) -> Exp s        -> Exp t
```

As is apparent from this definition, the typing rule for functions in $\lambda_{\to}$ can be expressed straightforwardly in terms of Haskell's typing rule for functions.

It is difficult in general to define recursive functions over higher-order terms.

```
data ExpF a where
  Lam   :: (a t1 -> a t2)           -> ExpF (a (t1 -> t2))
  App   :: a (t1 -> t2) -> a t1     -> ExpF (a t2)

  Pair  :: a t1 -> a t2             -> Expf (a (t1, t2))
  Fst   :: a (t1, t2)               -> ExpF (a t1)
  Snd   :: a (t1, t2)               -> ExpF (a t2)

  Num   :: Int                      -> ExpF (a Int)
  Prim  :: PrimOp -> a Int -> a Int -> ExpF (a Int)
  If0   :: a Int -> a t -> a t      -> ExpF (a t)

data Rec a b t = Roll (a (Rec a b t))  | Place (b t)

type Exp a t = Rec ExpF a t
```

Figure 2: Typeful, parametric representation of $\lambda_{\rightarrow}$

The problem comes from the fact that, in order to inspect the term "under a binder", one has to apply the corresponding meta-level function – and then, what information must be passed as argument? To alleviate this difficulty, it is useful to make use of an elimination form, commonly called a catamorphism (or iterator; we use the two terms interchangeably here, although they are given more specific meaning elsewhere [24]). A catamorphism encapsulates the traversal of a recursive structure; more precisely, it is a (higher-order) function that, given an elementary operation to perform on a single element, applies this operation to every element of the structure. (The most familiar instance of a catamorphism being the `fold` function over lists.)

In this work, we make use of Fegara and Sheard's catamorphism [5], over a parametric program representation encoded in Haskell [24]. In the remainder of this section, we briefly show how such an iterator is used and what modifications must be made to the (naive) program representation shown above, and illustrate its use with a simple example.

Figure 2 shows the representation we use. It differs from the naive representation in two ways:

1. It has been split into two types, `ExpF` and `Exp`, in order to make the recursive structure of the representation explicit. The type `ExpF` is the "prototype" representation, where the type argument `a` stands for the recursive form of the type. The recursive form, defined by the type `Exp`, is obtained by application of a sort of fixed-point operator, which is represented by the type constructor `Rec`. (You can ignore the data constructor `Place`, used internally by the iterator; see [5] if you are curious).

2. The representation is *parametric* in a type argument `a`, that is, a $\lambda_{\rightarrow}$ term of source type t is represented by a term of type $\forall$`a. Exp a t`, where `t` is the Haskell type that represents t. When applying the catamorphism, the type variable `a` is instantiated with the type that represents the information associated with a term (for instance, in the example of the the pretty-

```
xmapExpF :: (∀t. (a t -> b t, b t -> a t))
         -> (∀t. (ExpF (a t) -> ExpF (b t), ExpF (b t) -> ExpF (a t)))
cata  ::  (∀t. (ExpF (a t) -> a t)) -> (∀t. Exp a t -> a t)
iter  ::  (∀t. ExpF (b t) -> b t) -> (∀t. ((∀a. Exp a t) -> b t))

showAux :: ExpF ([String] -> String) -> ([String] -> String)
showAux (Num n) (v:vars) = show n
showAux (App x y) vars =
 "(" ++ (x vars) ++ " " ++ (y vars) ++ ")"
showAux (Lam z) (v:vars) =
 "(fn " ++ v ++ "= " ++ (z (const v) vars) ++ ")"
...

showE :: (∀a. Exp a t) -> String
showE e = iter showAux e vars
  where vars = ['a' .. 'z'] ++ ...
```

Figure 3: Pretty-printer implementation using Fegara and Sheard's iterator.

printer below, that information is the textual representation of the term represented as a string.)

Figure 3 shows the type of the iterator along with an example of its application. The internal functions `xmapExpF`, `cata`, and `iter` are taken from [24] and adapted to the case of a typed representation. The pretty-printer implementation consists of two functions: `showAux`, which shows an individual node of the syntax tree, and `showE`, which shows an entire tree and is obtained by application of the iterator.

Indeed, in our higher-order program representation, program variables are represented as Hasell variables, and thus have no identifiers associated with them. The pretty-printer assigns identifiers to variables as the traversal proceeds. The information associated with a term is its textual representation, which is parameterized by a list of identifiers; thus the type of terms `∀a. Exp a t` is instantiated as `Exp ([String] -> String) t`. In an imperative language, we would have simply used a *gensym* facility, but Haskell being side-effect-free, we have to thread a list of available identifiers in the display function.

3 CPS conversion

In this section we present the core contribution of this paper: an implementation of a CPS transformation where the type system of Haskell is used to encode the proof that this implementation correctly preserves types.

We proceed as follows. We first show the CPS conversion in its theoretical form; then define the typed representation of the target language λ_K; then show how to encode witnesses of type correspondence using existential types and GADTs; and finally show how the *functional dependency* between a type and its CPS form, a crucial point for completing the type correspondence proof, can also be encoded using GADTs.

$$
\begin{array}{rcl}
\mathcal{K}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int} \\
\mathcal{K}_{\text{type}}[\![\tau_1 \times \tau_2]\!] & = & \mathcal{K}_{\text{type}}[\![\tau_1]\!] \times \mathcal{K}_{\text{type}}[\![\tau_2]\!] \\
\mathcal{K}_{\text{type}}[\![\tau_1 \to \tau_2]\!] & = & (\tau_1 \times (\tau_2 \to 0)) \to 0 \\
\\
\mathcal{K}[\![x]\!]\ \kappa & = & \kappa\ x \\
\mathcal{K}[\![\lambda x{:}\tau_1.\, e : \tau_2]\!]\ \kappa & = & \kappa\ (\lambda(x, c){:}\mathcal{K}_{\text{type}}[\![\tau_1]\!] \times (\mathcal{K}_{\text{type}}[\![\tau_2]\!] \to 0).\, \mathcal{K}[\![e]\!]\ \ c) \\
\mathcal{K}[\![e_1\ e_2]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ x_1\ (x_2, \kappa))) \\
\mathcal{K}[\![(e_1, e_2)]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ \kappa\ (x_1, x_2))) \\
\mathcal{K}[\![\pi_i\ e]\!]\ \kappa & = & \mathcal{K}[\![e]\!]\ (\lambda x.\ \mathsf{let}\ x = \pi_i\ x\ \mathsf{in}\ \kappa\ x) \\
\mathcal{K}[\![i]\!]\ \kappa & = & \kappa\ i \\
\mathcal{K}[\![e_1\ p\ e_2]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ \mathsf{let}\ x_3 = x_1\ p\ x_2\ \mathsf{in}\ \kappa\ x_3)) \\
\mathcal{K}[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x.\ \mathsf{if0}\ x\ (\mathcal{K}[\![e_2]\!]\ \kappa)\ (\mathcal{K}[\![e_3]\!]\ \kappa)) \\
\\
\mathcal{K}_{\text{prog}}[\![e]\!] & = & \mathcal{K}[\![e]\!]\ \ (\lambda x.\ \mathsf{halt}\ x)
\end{array}
$$

Figure 4: CPS conversion

3.1 The theory

Conversion to continuation-passing style (CPS) names all intermediate computational 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 applying 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. The target language of the CPS conversion, here called λ_K has the following syntax:

$$
\begin{array}{ll}
(\textit{types}) & \tau ::= \tau \to 0 \mid \mathsf{int} \mid \tau_1 \times \tau_2 \\
(\textit{type env}) & \Gamma ::= \bullet \mid \Gamma, x{:}\tau \\
(\textit{values}) & v ::= x \mid i \mid \lambda x{:}\tau\ .\ e \mid (v_1, v_2) \\
(\textit{primops}) & p ::= + \mid - \mid \times \\
(\textit{exps}) & e ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \pi_i\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \\
 & \quad \mid v_1\ v_2 \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v
\end{array}
$$

It differs from $\lambda_\to$ in that its syntax is split into two syntactic categories of expressions and values. Values represent those things that can be bound to a variable: either another variable, or the introduction forms for functions, integers or pairs. Expressions consist of a list of declarations (introduced by let forms), followed by either a function application, a conditional expression, or the special form halt, which indicates the final "answer" produced by the program. The fact that a function does not return to the caller is reflected in its type as $\tau \to 0$.

Figure 4 shows the CPS conversion itself. It is defined in three functions. The main function, $\mathcal{K}[\![-]\!]\ \kappa$, transforms a $\lambda_\to$ expression in its CPS form expression, given a continuation κ. The function $\mathcal{K}_{\text{type}}[\![-]\!]$, for each type in $\lambda_\to$, gives the

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_K x : \tau} \qquad \frac{\Gamma, x{:}\tau \vdash_K e}{\Gamma \vdash_K \lambda x{:}\tau.\, e : \tau \to 0} \qquad \frac{}{\Gamma \vdash_K i : \mathsf{int}} \qquad \frac{\forall i \;.\; \Gamma \vdash_K v_i : \tau_i}{\Gamma \vdash_K (v_1, v_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash_K v_1 : \tau \to 0 \quad \Gamma \vdash_K v_2 : \tau}{\Gamma \vdash_K v_1\ v_2} \qquad \frac{\Gamma \vdash_K v : \tau \quad \Gamma, x{:}\tau \vdash_K e}{\Gamma \vdash_K \mathsf{let}\ x = v\ \mathsf{in}\ e}$$

$$\frac{\Gamma \vdash_K v : \tau_1 \times \tau_2 \quad \Gamma, x{:}\tau_i \vdash_K e}{\Gamma \vdash_K \mathsf{let}\ x = \pi_i\ v\ \mathsf{in}\ e} \qquad \frac{\Gamma \vdash_K v_1 : \mathsf{int} \quad \Gamma \vdash_K v_2 : \mathsf{int} \quad \Gamma, x{:}\mathsf{int} \vdash_K e}{\Gamma \vdash_K \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e}$$

$$\frac{\Gamma \vdash_K v : \mathsf{int} \quad \Gamma \vdash_K e_1 \quad \Gamma \vdash_K e_2}{\Gamma \vdash_K \mathsf{if0}\ v\ e_1\ e_2} \qquad \frac{\Gamma \vdash_K v : \tau}{\Gamma \vdash_K \mathsf{halt}\ v}$$

Figure 5: Typing rules for values and expressions of λ_K

corresponding type in λ_K. (Note that this function is used to convert the type annotations in the case $\mathcal{K}[\![\lambda x{:}\tau_1.\, e : \tau_2]\!]$). Finally, $\mathcal{K}_{\text{prog}}[\![-]\!]$ converts an entire program by arranging for the final result to be passed to the special form halt.

3.2 Type preservation

The static semantics shown in Fig. 5 defines two typing judgments: $\Gamma \vdash_K v : \tau$ assigns type τ to value v; while $\Gamma \vdash_K e$ asserts that expression e is well typed.

In its simplest form, type preservation states that if a program is well-typed in $\lambda_{\to}$, then the program after CPS conversion will also be well-typed:

Theorem 3.1 *(CPS type preservation) If* $\bullet \vdash e : \tau$*, then* $\bullet \vdash_K \mathcal{K}_{prog}[\![e]\!]$.

In order to prove the above theorem, it is useful to prove a stronger property that establishes the correspondence between the types in $\lambda_{\to}$ and those in λ_K. We can state this correspondence formally as follows:

Theorem 3.2 *($\lambda_{\to}$–λ_K type correspondence) If* $\bullet \vdash e : \tau$*, then* $\bullet \vdash_K \lambda c.\, \mathcal{K}[\![e]\!]\ c : (\mathcal{K}_{type}[\![\tau]\!] \to 0) \to 0$.

Note that the expression in CPS is "wrapped" into a λ-abstraction and thus turned into a value, so that it can be given a type.

3.3 Program representation

Figure 6 shows the typed representation of λ_K. Ideally, we would like to define two mutually recursive types, `ValK` and `ExpK`, representing the syntactic categories of values and expressions, respectively. However, our fixed point operator (`Rec`, see Fig. 2) can only be applied to a single type, so instead we use the same type for the two syntactic categories. (Alternatively, one might prefer to extend the recursion scheme to the case of two or more types, but we do not attempt this here.)

```
data V t
data ExpKF a where
    -- values
    KVnum     :: Int                          -> ExpKF (a (V Int))
    KVlam     :: (a (V s) -> a Z)             -> ExpKF (a (V (s -> Z)))
    KVpair    :: a (V s) -> a (V t)           -> ExpKF (a (V (s, t)))
    -- expressions
    Klet_val  :: a (V t) -> (a (V t) -> a Z)         -> ExpKF (a Z)
    Klet_fst  :: a (V (t1, t2)) -> (a (V t1) -> a Z) -> ExpKF (a Z)
    Klet_snd  :: a (V (t1, t2)) -> (a (V t2) -> a Z) -> ExpKF (a Z)
    Klet_prim :: PrimOp -> a (V Int) -> a (V Int) -> (a (V Int) -> a Z)
                                                     -> ExpKF (a Z)
    Kapp      :: a (V (s -> Z)) -> a (V s)           -> ExpKF (a Z)
    Kif0      :: a (V Int) -> a Z -> a Z             -> ExpKF (a Z)
    Khalt     :: a (V Int)                           -> ExpKF (a Z)

type ValK a t = Rec ExpKF a (V t)
type ExpK a   = Rec ExpKF a Z
```

Figure 6: Typeful representation of λ_K

The distinction between expressions and values is actually not lost: we take advantage of the GADTs to recover this distinction by encoding the corresponding syntactic constraints as type constraints: values have source type `V t` whereas expressions have source type `Z`, so types statically enforce that constructors for values cannot appear where an expression is expected and vice versa.

3.4 Proving type correspondence

At first approximation, by applying the Curry-Howard isomorphism, the type correspondence property of the CPS transform (Theorem 3.2) might be reflected in the type of its implementation in this way:

```
cps :: (∀a. Exp a t) -> (∀a.ValK a K_type⟦t⟧ -> ExpK a) -> (∀a. ExpK a)
```

Here, indeed, we abuse notation by using $\mathcal{K}_{\text{type}}[\![-]\!]$ in a Haskell type expression – we cannot express $\mathcal{K}_{\text{type}}[\![-]\!]$ directly since Haskell lacks intensional type analysis at the level of types. To circumvent the problem, we encode a *proof* of the correspondence between `t` and $\mathcal{K}_{\text{type}}[\![\texttt{t}]\!]$. That is, we instead type `cps` as follows:

```
cps :: (∀a. Exp a t)
      -> ∃cps_t. (CpsForm t cps_t,
                  (∀a. (ValK a cps_t -> ExpK a) -> ExpK a))
```

where a value of type `CpsForm t cps_t` represents a proof that `cps_t` $= \mathcal{K}_{\text{type}}[\![\texttt{t}]\!]$. Such a proof is encoded in a GADT whose data constructors only permit the creation of valid associations between a type in the source language and its corresponding type in CPS form:

```
data CpsForm t cps_t where
  CpsInt  :: CpsForm Int Int
  CpsPair :: CpsForm s cps_s -> CpsForm t cps_t
               -> CpsForm (s, t) (cps_s, cps_t)
  CpsFun  :: CpsForm s cps_s -> CpsForm t cps_t
               -> CpsForm (s -> t) ((cps_s, cps_t -> Z) -> Z)
```

Now, since we use HOAS, we have to structure the CPS transformation slightly differently: we will define a function that performs CPS conversion of a single node, and apply the iterator to this function in order to obtain a function that converts an entire program (like we did in the pretty-printer example of Sec. 2.2.) The type of the function performing CPS conversion of an individual node has the following type:

```
cpsAux :: ∀a. ExpF (CPS a t) -> CPS a t
```

where `CPS a t` represents the CPS-converted form of an expression of source type `t`, and is an abbreviation whose meaning is defined as follows:

```
type CPS a t =
    ∃cps_t. (CpsForm t cps_t,
             ((ValK a cps_t -> ExpK a) -> ExpK a))
```

To illustrate the technique, the case that CPS-converts a pair construction term (a, b) is implemented as follows:

```
cpsAux (Pair (a::CPS a s) (b::CPS a t)) =
  case (a, b) of
    ((s_cps_s, cps_a), (t_cps_t, cps_b)) ->
      ((CpsPair s_cps_s t_cps_t),
       (λk -> (cps_a (λv1 -> cps_b (λv2 -> k (pairK v1 v2))))))
```

As can be seen from this example, the code follows the structure of an inductive proof, where the CPS transformation and its proof of type-preservation are interlaced.

Finally, the main function of the CPS transformation:

```
cpsProg :: (∀a. Exp a t) -> (∀a. ExpK a)
```

is obtained by applying the iterator to the function `cpsAux`. (Since it implements $\mathcal{K}_{\text{prog}}[\![-]\!]$, its type does not reflect the type correspondence property, only type preservation.)

3.5 Functional dependency

In some places of the type correspondence proof, we need to use the fact that the CPS form of a given type in $\lambda_{\rightarrow}$ is unique, that is:

Theorem 3.3 *(Uniqueness of CPS form) If* $\mathcal{K}_{type}[\![\tau]\!] = \tau_K$ *and* $\mathcal{K}_{type}[\![\tau]\!] = \tau'_K$, *then* $\tau_K = \tau'_K$.

We refer to this fact as a *functional dependency* between a type τ and its CPS form $\mathcal{K}_{\text{type}}[\![\tau]\!]$, in the sense of [7]. By the Curry-Howard isomorphism we can encode this theorem as a Haskell function. First, we encode type equality using a GADT:

```
data Equal a b where
    Eq_refl :: Equal a a
```

whose only introduction form accounts for reflexivity. Then Theorem 3.3 is proved as follows:

```
cpsUnique :: CpsForm t cps_t -> CpsForm t cps_t' -> Equal cps_t cps_t'
cpsUnique CpsInt CpsInt = Eq_refl
cpsUnique (CpsFun (s_cps_s::CpsForm s cps_s)
                  (t_cps_t::CpsForm t cps_t))
          (CpsFun (s_cps_s'::CpsForm s cps_s')
                  (t_cps_t'::CpsForm t cps_t'))
  = case cpsUnique s_cps_s s_cps_s' of
      (Eq_refl::Equal cps_s cps_s') ->
        case cpsUnique t_cps_t t_cps_t' of
          (Eq_refl::Equal cps_t cps_t') ->
             Eq_refl
```

We make use of this theorem, for instance, in the case of function application where we need to use the fact that the CPS form of the argument (`e2`) matches the type expected by the CPS-converted function (`e1`):

```
cpsAux (App (e1::CPS a (s->t))
            (e2::CPS a s)) =
case e1 of
  (CpsFun s_cps_s t_cps_t, m1) ->
    case e2 of (s_cps_s', m2) ->
      case cpsUnique s_cps_s s_cps_s' of Eq_refl ->
        (t_cps_t,
         (λk -> m1 (λr1 -> m2 (λr2 -> appK r1 (pairK r2 (contK k))))))
```

4 Fine points

We discuss here some differences between the previous section and the code we actually use; the problem of unsoundness of our proofs and a way we tried to solve it; as well as how we solve the problem of constructing the HOAS terms, which we have for now conveniently skipped.

4.1 The CPS conversion of Danvy and Filinski

Danvy and Filinski's one-pass CPS conversion [4], where administrative redexes are reduced on-the-fly, can be conveniently expressed using an iterator over a HOAS, as was illustrated in Washburn and Weirich's paper [24]. The essential difference with the conversion shown above is reflected in the representation of a CPS-converted term which, in our setting, would be as follows:

```
type CPS a t =
  ∃cps_t. (CpsForm t cps_t,
           ((ValK a cps_t -> ExpK a) -> ExpK a),     cps-meta
           ((ValK a (cps_t -> Z)) -> ExpK a))        cps-obj
```

A term in CPS is now represented by both (1) a term *cps-meta* parameterized by a meta-level continuation, as before, and (2) a term *cps-obj* parameterized by an object-level continuation, that is, a value of source type `(cps_t -> Z)`. Thus the CPS conversion of a term simultaneously defines these two forms.

We have treated type preservation in the case of the basic CPS transformation in order to simplify the presentation; our compiler actually implements Danvy and Filinski's CPS conversion. The type preservation proof extends to this case without particular difficulty.

4.2 (Un)soundness

One concern with our approach is that the type-preservation proof is encoded in an unsound logic. That is, one can trivially encode a "proof" of type correspondence between any two types `s` and `t` (that is, a value of type `CpsForm s t`) as a non-terminating Haskell term.

At any rate, the compiler could be made to traverse the type-preservation proof after the fact to verify that it is indeed complete – this pass would simply diverge in the event of an incorrect proof.

Of course, one must be careful not to introduce non-terminating terms when developing a proof. The risk is slight, however, the presence of such terms being fairly manifest, and given the fact that *we* are writing the proof. That is, we are not in a PCC setting where the possibility of a malicious adversary exploiting any loop-holes of our logic is a prime consideration. Here, the construction of witnesses is merely a device to verify our intuition. For that purpose, we believe that the degree of confidence provided by our technique is reasonable, although we clearly hope to find something better.

4.3 Haskell type classes

Before resorting to manipulating explicit proofs in an unsound logic, we tried another approach that relied on multi-parameter type classes. This approach initially seemed much more promising and elegant.

The intended use of type classes in Haskell is to control *ad-hoc* polymorphism. A type class can be seen as a predicate asserting the existence of a set of functions defined over that type, the implementation of these functions being provided as part of an *instance* declaration. For example, the `Show` class states the existence of a `show` function of type `t -> String`, defined for each type `t` that is a member of the class. In the Haskell 98 standard, a type class may involve only a single type argument. However, a common extension supported by Haskell compilers permits the definition of multi-parameter type classes, which extend the notion of predicates over types to that of *relations* among types.

Thus, one can declare a type class that represents a relation between types in $\lambda_{\rightarrow}$ and type in λ_K as follows:

```
class CpsForm t cps_t
```

The relation is defined as a set of instance declarations as follows:

```
instance CpsForm Int Int
instance (CpsForm s cps_s, CpsForm t cps_t)
   => CpsForm (s, t) (cps_s, cps_t)
instance (CpsForm s cps_s, CpsForm t cps_t)
   => CpsForm (s -> t) ((cps_s, cps_t -> Z) -> Z)
```

This set of instance of declarations can be viewed as (static) type-level logic programming. Each instance declaration can be read as an inference rule: the first rule is an axiom that states the CPS form of `Int` is `Int`, the second rule states that the CPS form of `(s, t)` is `(cps_s, cps_t)`, provided `cps_s` is the CPS form of `s` and `cps_t` is that of `t`, and similarly for the third rule. Finally, we can express the fact that the relation is a function with an additional clause (a *functional dependency*) to the class declaration as follows:

```
class CpsForm t cps_t | t -> cps_t
```

Now, making use of the type class, we can express type preservation as follows. We'd keep the type of `cpsAux` as before, that is:

```
cpsAux :: ExpF (CPS a t) -> CPS a t
```

but the type synonym `CPS a t` would now stand for the following (existential) type:

```
type CPS a t =
    ∃cps_t. CpsForm t cps_t =>
              (ValK a cps_t -> ExpK a) -> ExpK a
```

Unfortunately, in practice, this scheme doesn't take us very far. GHC isn't currently able to type-check this code, even though it appears logically correct. For this to work, we'd expect the type checker to apply the functinnal dependency and instance declarations to identify the unique type `cps_t` given `t` and to use this information as input for GADT type refinement. But such precise interaction between functional dependencies and GADTs isn't currently present in GHC. The situation may change in the future, if for instance a new internal representation is adopted in GHC [22].

It is worth noting that associated types [1] may provide an attractive alternative to functional dependencies. But in the absence of a robust implementation of associated types, it is unclear at the moment whether we would face the same difficulties as with type classes.

4.4 Construction of higher-order terms

The compiler front-end performs a lexical and syntactic analysis and produces an abstract syntax tree. Here, the abstract syntax tree is a term in higher-order abstract syntax. Constructing an efficient representation of such higher-order terms is the subject of some concern. To illustrate, suppose that one attempts to construct a parser that directly produces a higher-order representation; then one invariably ends up writing a parser having essentially this form:

```
parse ... = case ... of
              ... -> Lam (λx -> ... (parse x ...) ... )
              ...
```

The problem is that the body of the function may indeed refer to the newly bound variable (`x`), so the variable has to be passed as argument to `parse` in the recursive call. Thus the resulting syntax tree contains a call to `parse` under every `Lam` node, with dramatic consequences on the compiler's performance.

Fortunately, there is a simple solution to this problem. A higher-order representation can be constructed by *meta-programming*, that is, by using an extension of Haskell through which fragments of Haskell code can be manipulated under program control. We make use of Template Haskell [20], a meta-programming facility now included in GHC.

In our compiler, we use a parser producing a first-order abstract syntax, and then turn it into a HOAS term using Template Haskell. The first-order syntax trees are represented in a conventional manner:

```
data AST where
  Fvar :: Ident -> AST
  Flam :: Ident -> AST -> AST
  Fapp :: AST -> AST -> AST
  ...
```

where `Ident` is a type for identifiers. We define a Template Haskell function `lift` that turns this representation into HOAS:

```
lift :: AST -> ExpQ
lift (Fvar x)     = varE (mkName x)
lift (Flam x t b) = [| Lam $(lam1E (varP (mkName x)) (lift b)) |]
lift (Fapp a b)   = [| App $(lift a) $(lift b) |]
...
```

The type `ExpQ` is a type defined by Template Haskell for representing Haskell expressions. The code in semantic brackets (`[|−|]`) represents a quoted expression, and the form `$(−)` is used to escape from the quotes (much in the manner of Scheme's `quasiquote` and `unquote`.)

Now, we can apply the above function with the special form `$(lift ast)`. Thus, the main function of the compiler follows this structure:

```
compile :: ProgramText -> Assembly
compile program_text =
   let ast = parse program_text
       exp = $(lift ast)
   in (generate_code . closure_conversion . cps_conversion) exp
```

In essence, `lift` rewrites the source program in Haskell, in terms of the constructors that define our HOAS representation. If the resutling Haskell code is well-typed, then so is the source program – thus we also get a source-level type-checker for free, courtesy of GHC.

5 Related work

There has been a lot of work on typed intermediate languages, beginning with the TIL [23] and FLINT [17, 16] work, originally motivated by the optimizations opportunities offered by the extra type information. [12] introduced the idea of Proof-Carrying Code, making it desirable to propagate type information even further than the early optimization stages, as done in in [11].

In [19], Shao et al. show a low-level typed intermediate languages 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.

In [13], Emir Pasalic develops 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 deBruijn indices and relies on type equality proofs in Haskell.

In [2], Chiyan Chen et al. 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 deBruijn indices, and their implementation language is more experimental. In a similar vein, Linger and Sheard [10] show a CPS transform over a GADT-based representation with deBruijn 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.

In [9], Leroy shows a backend of a compiler written in the Coq proof assistant, and whose correctness proof is completely formalized. He uses a language whose type systems is much more powerful than ours, but whose computational language is more restrictive.

In [5], Fegaras and Sheard show how to handle higher-order abstract syntax, and in [24], Washburn and Weirich show 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.

GADTs were introduced many times under many different names [25, 3, 21]. Their interaction with type classes is a known problem in GHC and a possible solution was proposed in [22].

6 Discussion and future work

The use of HOAS raises concerns about the performance of the compiler. There is a question whether it will incur a significant amount of repeated work, as would have been the case in the parser had we not used Template Haskell. The answer wholly depends on the structure of the compiler: if it is streamlined to the point that each intermediate representation is used only once, then performance won't suffer much. But repeated analysis phases over the same intermediate representation would clearly result in repeated work. In this case, we'd simply use Template Haskell again to "flatten" the representation after certain phases and thus recover viable performance.

Of course we intend to add many more compilation phases, such as closure conversion, optimization, register allocation, to make it a more realistic compiler. Closure conversion in particular offers a greater challenge than CPS since it is somewhat more intensive w.r.t. program analysis. The type of a code fragment (at least locally, i.e. within a closure) depends on its free variables. This mean some program analysis will have to take place statically in order to be reflected in Haskell's type system.

We also intend to make our source language more powerful by adding features such as parametric polymorphism and recursive types.

Also we hope to find some clean way to move the unsound term-level manipulation of proofs to the sound 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 [6]: 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.

6.1 Conclusion

We have shown how to write some parts of a compiler using GADTs such that the type system of the language in which the compiler is written can automatically verify that the compiler properly preserves the types of its programs. We have specifically shown how to write the CPS conversion and the conversion from an untyped representation to a typed representation.

As part of this, we have shown how to integrate generalized algebraic data types with Washburn and Weirich's technique to encode higher-order abstract syntax in a Haskell-like language. We have also shown how to use Template Haskell to leverage Haskell's type checker to do our type checking for us.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 1–13, New York, NY, USA, 2005. ACM Press.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] Mark P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–244, 2000.

[8] Xavier Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages*, pages 177–188, January 1992.

[9] 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.

[10] Nathan Linger and Tim Sheard. Programming with static invariants in omega. Unpublished, 2004.

[11] 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.

[12] 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.

[13] 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.

[14] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press.

[15] Zhong Shao. Flexible representation analysis. In *International Conference on Functional Programming*, pages 85–98. ACM Press, June 1997.

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

[17] 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.

[18] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323. ACM Press, September 1998.

[19] 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.

[20] 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.

[21] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages*, Cork, July 2004.

[22] Martin Sulzmann, Manuel M. T. Chakravarty, and Simon Peyton Jones. System F with type equality coercions. Submitted to ICFP'06.

[23] 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.

[24] 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.

[25] 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.