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 in the types important invariants of the program. This allows type-based program verification and is a promising compromise between plain old types and full blown Hoare logic proofs.

With the introduction of GADTs in GHC, such dependent typing is finally available in an industry-quality implementation, making it possible to consider its use in large scale programs. One of the most common examples of GADTs (after the proverbial $List\ \alpha\ n$) is to annotate a mini language with its typing rules, so we decided to develop the toy example to a complete compiler whose main property is that the GHC type checker verifies mechanically that each phase of the compiler properly preserves types.

The use of typed intermediate languages is fairly common nowadays, but is normally limited to *testing* that the generated code is properly typed, whereas we get to *verify* formally that our compiler indeed preserves typing. This has already been done to some extent, but in proof assistants or experimental languages, and usually limited to a particular phase (typically the CPS transformation), or else with a much more ambitious goal which requires much more extensive proof annotations.

Other than guaranteeing that types are preserved, our compiler has the following unusual characteristics: it uses Template Haskell to get a type checker for free; it uses de Bruijn indices in its closure conversion phase but higher-order abstract syntax (HOAS) in its CPS conversion phase, so it includes a conversion to and from HOAS.

General Terms

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

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 exploring more powerful type systems 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 [Leroy, 1992, Tarditi et al., 1996, Shao, 1997a], sanity checks to help catch compiler errors, and more recently to help construct proofs that the generated code verifies some safety properties [Morrisett et al., 1999, Hamid et al., 2002]. Typically those compilers represent the source level types 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:

- It amounts to *testing* the compiler, thus bugs can lurk, undetected.
- A detected type error, reported as an "internal compiler error", will surely annoy the user, who generally holds no responsibility for what went wrong.
- It incurs additional work, obviously, which can slow down the compiler.
- Errors are only detected when we run the type checker, but running it as often as possible slows down our compiler even more.

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.

We believe that type preservation by a compiler is the perfect example of the kind of properties that type systems of the future should allow programmers to conveniently express and verify. Others (e.g. [Chen and Xi, 2003]) have used typeful program representations to statically enforce type preservation, but as far as we know, the work presented here is the first attempt to do so using a language so widely used and well supported as Haskell, for which an industrial strength compiler is available. Also, to our knowledge, this is the first time such a technique is applied to closure conversion and hoisting.

This work follows a similar goal to the one of [Leroy, 2006, Blazy et al., 2006], 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.

In an earlier article [Guillemette and Monnier, 2006], we presented the CPS phase of our compiler, which used a higher order abstract syntax (HOAS) [Pfenning and Elliot, 1988] representation of terms to render the type preservation proof easier. In this article we present the closure conversion and function hoisting phases, both of which use a first order representation of terms, using de Bruijn indices. We found a first order representation to be easier to use for closure conversion.

Our main contributions are the following:

- We show type-preserving CPS and closure conversions as well as a function hoisting phase, all written in Haskell with GHC extensions (mainly GADTs) and where the GHC type checker 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, once using HOAS and once using de Bruijn indices.
- We use higher-order abstract syntax (HOAS) in our intermediate representation, following [Washburn and Weirich, 2003], and we show how to combine this technique with GADTs and how to build such terms using Template Haskell [Sheard and Jones, 2002].
- Our combination of GADTs with meta programming lets GHC do the type checking of our source programs for us.
- We additionally show a type preserving conversion from strongly typed HOAS terms into strongly typed first order terms using de Bruijn indices.

Overview The source language we are compiling is a simple call-by-value functional language currently limited to monomorphic types. The general compilation strategy follows the one used in [Morrisett et al., 1999]: after parsing and type checking we first fix the order of evaluation and make stack manipulation explicit using a CPS conversion; then make the manipulation of closures explicit using a closure conversion phase; then hoist all the functions (which are now all closed) to the top-level; after that should come register allocation and code generation, although we have not implemented that part yet.

To let the Haskell type checker verify that we preserve types, we need to make our types visible to Haskell's type checker. The way we do that is that instead of representing each intermediate language using abstract data types like `Exp`, we use GADTs that let us encode the typing rules of our languages and give our terms types such as `Exp t` where t represents the source-level type of the expression. Then we can express the type preservation property of each phase by giving it a type such as $\forall$`t.Exp`$_{in}$ `t -> Exp`$_{out}$ `t`. To be more precise since the type of the output code is related but may be different from the type of the output code, the type of a phase will be more like $\forall$`t.Exp`$_{in}$ `t -> Exp`$_{out}$ `(`f `t)` where f is a type-level function that expresses how source-level types are changed by this phase.

Outline This article is structured as follows: Section 2 briefly presents the techniques used to represent our intermediate terms. Section 3 gives an overview of the compiler. Then we show each phase of the compiler in more details in Sec. 4, 5, 6, 7, and 8. Section 9 concludes with related and future work.

2. Background

In this section we briefly introduce the techniques we use in the representation of our intermediate terms, namely GADTs, HOAS, and de Bruijn indices. GADTs provide a limited form of dependent typing sufficient to encode the typing rules of our source language, so they are our main tool to express the typing preservation property. HOAS and de Bruijn are two standard ways to represent variable bindings, both of which are used in our compiler.

Generalized algebraic datatypes Generalized algebraic datatypes (GADTs) [Xi et al., 2003, Cheney and Hinze, 2003] 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 primarily use GADTs to represent abstract syntax trees, and use these type annotations to track the source-level type of expressions. For example, consider some common typing rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}}$$

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

```
data Exp where
  Cst :: Int -> Exp
  App :: Exp -> Exp -> Exp
  ...
```

where the source types 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 typing rule:

```
data Exp t where
  Cst :: Exp Int
  App :: 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 (`t1 -> t2`) to represent 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 t1 t2`, to achieve the same effect.

While this example captures the essential feature of GADTs we need, there remain non-trivial decisions to be made concerning the way we use such annotations to track the type of binders as they are introduced in syntactic forms like λ or let.

HOAS. Consider the 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, this typing rule would be encoded as follows:

```
data Exp t where
  Let :: Exp t1 -> (Exp t1 -> Exp t2) -> Exp t2
  ...
```

that is, binders in source programs would be represented by Haskell binders – and thus there is not need for an explicit introduction form for variable occurrences. 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. The technique is particularly concise, but its simplicity has a cost: explicit constraints on the type context of a term cannot be expressed. For instance, we cannot express the fact that a term is closed.

De Bruijn indices. In contrast to HOAS, a first-order representation introduces variables explicitly. With de Bruijn indices, as with HOAS, variables 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 is a list of types that represents the expression's type context:

```
data Exp ts t where
  Bvar :: Index ts i 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 i t` represents a de Bruijn index with index value `i`, whose type is `t` within the type environment `ts`. Such indices are represented with type-annotated Peano numbers:

```
data Index ts i t where
  I0 :: Index (t, ts) Tzero t
  Is :: Index ts n t -> Index (t0, ts) (Tsucc n) t
```

where `Tzero` and `Tsucc` reify the natural numbers as types. Note that individual indices are polymorphic in `ts` and `t`, and assume a particular type given a particular type context `ts`.

3. Overview

As mentioned in the introduction, the source language we are compiling is a simple call-by-value functional language currently limited to monomorphic types. The general compilation strategy follows the one used in [Morrisett et al., 1999]. The general structure of our compiler is as follows:

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

The first phase infers types for all subterms of the source program, and all the subsequent ones are then careful to preserve them. In this section we will briefly show what each of those phases does to the code and the types.

3.1 Type checking: `AST -> ∃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 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 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. Constructing an efficient HOAS representation generally requires some form of meta programming, so we use Template Haskell for that phase, which gives us the type checker for free.

3.2 CPS conversion: `Exp t -> 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 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. Additionally a special form halt is used to indicate the final "answer" produced by the program. For example:

$$\begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad b = 32\\ \quad c = 24\\ \quad c2f = \lambda x\ .\ a \cdot x\ +\ b\\ \mathsf{in}\ c2f\ c\end{array} \quad \overset{CPS}{\Longrightarrow} \quad \begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad b = 32\\ \quad c = 24\\ \quad c2b = \lambda\langle x, k\rangle\ .\\ \qquad \mathsf{let}\ v_0 = a \cdot x\\ \qquad\quad v_1 = v_0 + b\\ \qquad \mathsf{in}\ k\ v_1\\ \mathsf{in}\ c2b\ \langle c, \lambda v\ .\ \mathsf{halt}\ v\rangle\end{array}$$

For an input expression of type `Exp t` the output type should be `ExpK (cps t)` where `cps` is a type-level function that describes the way types are modified by this phase: mostly input types of the form t_1 `->` t_2 are mapped to `(`t_1' `*` `(`t_2' `-> Z)) -> Z` where `Z` is the void type.

3.3 Conversion to de Bruijn: `ExpK t -> ExpKB ts t`

While HOAS is convenient for the CPS conversion, it cannot be used (or 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 codets. 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.

3.4 Closure conv: `ExpKB ts t -> 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:

$$\begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad b = 32\\ \quad c = 24\\ \quad c2f = \mathsf{closure}\ \lambda\langle\langle x, k\rangle, env\rangle\ .\ \mathsf{let}\ v_0 = env.0 \cdot x\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad v_1 = v_0 + env.1\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \langle k_f, k_{env}\rangle = \mathsf{open}\ k\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ k_f\ \langle v_1, k_{env}\rangle\\ \qquad\qquad\qquad\quad \langle a, b\rangle\\ \quad \langle c2f_f, c2f_{env}\rangle = \mathsf{open}\ c2f\\ \mathsf{in}\ c2b_f\ \langle\langle c, \mathsf{closure}\ (\lambda\langle v, env\rangle\ .\ \mathsf{halt}\ v)\ \langle\rangle\rangle, c2f_{env}\rangle\end{array}$$

Note that usually (as in [Morrisett et al., 1999]) closures are represented as existential packages, whereas we use special purpose constructs closure and open specifically to avoid introducing general existential in our language. We intend to add polymorphism and existential types to our source language, but this is a non-trivial future extension.

3.5 Hoisting: `ExpC ts t -> ExpH ts t`

After closure conversion, λ-abstractions are closed and can be moved to the top level. This phase is conceptually trivial, but a bit less so from a typing point of view which is why it also deserves a section of its own. The previous example after hoisting will look as

follows:

$$
\begin{array}{l}
\mathsf{let}\ \ell_0 = \lambda\langle\langle x, k\rangle, env\rangle\ .\ \mathsf{let}\ v_0 = env.0 \cdot x \\
\qquad\qquad\qquad\qquad\qquad\quad v_1 = v_0 + env.1 \\
\qquad\qquad\qquad\qquad\qquad\quad \langle k_f, k_{env}\rangle = \mathsf{open}\ k \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ k_f\ \langle v_1, k_{env}\rangle \\
\quad \ell_1 = \lambda\langle v, env\rangle\ .\ \mathsf{halt}\ v \\
\quad a = 1.8 \\
\quad b = 32 \\
\quad c = 24 \\
\quad c2f = \mathsf{closure}\ \ell_0\ \langle a, b\rangle \\
\quad \langle c2f_f, c2f_{env}\rangle = \mathsf{open}\ c2f \\
\mathsf{in}\ c2b_f\ \langle\langle c, \mathsf{closure}\ \ell_1\ \langle\rangle\rangle, c2f_{env}\rangle
\end{array}
$$

4. Type checking

Having the front-end produce HOAS raises the issue of residual redexes in the program representation. That is, a direct implementation leads to things like recursive calls to the parser hidden inside closures of functional arguments to constructors with functional arguments, like those for λ or let – with dramatic consequences on performance. This motivates the use of Template Haskell, a meta-programming facility for Haskell bundled with GHC, to generate a fresh HOAS representation to feed into subsequent phases. By so doing, we also get a source-level type-checker for free.

Our compiler employs a conventional parser producing first-order abstract syntax, and then "lifts" the program to HOAS through a Template Haskell function. The latter has type:

```
lift :: AST -> ExpQ
```

where `AST` is the first-order representation, and `ExpQ` is the Template Haskell type for representing Haskell expressions. Special syntax is provided that renders the implementation of `lift` similar to Scheme code with quasi-quotations. The main driver of the compiler has the form:

```
compile program_text =
   let ast = parse program_text
       exp = $(lift ast)
   in (hoist . cc . toB . cps) exp
```

where `$(−)` is a special form for applying Template Haskell functions. In essence, `lift` rewrites the source program in Haskell, in terms of the constructors that define our HOAS representation. If the resulting Haskell code is well-typed, then so is the source program.

5. CPS conversion

CPS conversion is relatively straightforward and allows us to demonstrate the basic elements that constitute the approach taken in all transformation phases, based around the pairing of GADT-encoded type correspondence witnesses along with the produced code. It is also a nice showcase of the elegance of HOAS. In comparison to de Bruijn indices as used in the closure conversion and hoisting phases, HOAS relieves us from any index-mangling code that tend to clutter the code with unwanted detail. We can even obtain a lightly optimizing CPS transform (eliminating administrative redexes on-the-fly) with very little effort.

The source and target languages of the CPS conversion are formalized in Fig. 1. The source language ($\lambda_{\to}$) is a simply typed, call-by-value λ-calculus, with a non-recursive let-form and integers as a base type. Its static and dynamic semantics are standard and are not reproduced here. However we will henceforth refer to a typing judgement $\Gamma \vdash e : \tau$ over $\lambda_{\to}$ expressions, assuming standard definitions.

The target language ($\lambda_{\mathcal{K}}$) 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 $\tau \to 0$. We will refer to typing judgements $\Gamma \vdash_{\mathcal{K}} e$ over $\lambda_{\mathcal{K}}$ expressions and $\Gamma \vdash_{\mathcal{K}} v : \tau$ over $\lambda_{\mathcal{K}}$ values, again assuming standard definitions.

Source language ($\lambda_{\to}$)

$$
\begin{array}{lll}
(\textit{types}) & \tau & ::= \tau_1 \to \tau_2 \mid \mathsf{int} \\
(\textit{type env}) & \Gamma & ::= \bullet \mid \Gamma, x{:}\tau \\
(\textit{primops}) & p & ::= + \mid - \mid \cdot \\
(\textit{exps}) & e & ::= x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \lambda x\ .\ e \mid e_1\ e_2 \mid n \\
 & & \quad \mid e_1\ p\ e_2
\end{array}
$$

CPS language ($\lambda_{\mathcal{K}}$)

$$
\begin{array}{lll}
(\textit{types}) & \tau & ::= \tau \to 0 \mid \tau_1 \times \cdots \times \tau_n \mid \mathsf{int} \\
(\textit{type env}) & \Gamma & ::= \bullet \mid \Gamma, x{:}\tau \\
(\textit{values}) & e & ::= x \mid \lambda x\ .\ e \mid \langle e_0, \ldots, e_{n-1}\rangle \mid e.i \\
(\textit{exps}) & e & ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \mid v_1\ v_2 \\
 & & \quad \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v
\end{array}
$$

Figure 1. Source and target languages of CPS conversion

$$
\begin{array}{rcl}
\mathcal{K}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int} \\
\mathcal{K}_{\text{type}}[\![\tau_1 \to \tau_2]\!] & = & (\tau_1 \times (\tau_2 \to 0)) \to 0 \\
\\
\mathcal{K}[\![x]\!]\ \kappa & = & \kappa\ x \\
\mathcal{K}[\![\lambda x\ .\ e]\!]\ \kappa & = & \kappa\ (\lambda\langle x, c\rangle\ .\ \mathcal{K}[\![e]\!]\ \ c) \\
\mathcal{K}[\![e_1\ e_2]\!]\ \kappa & = & \mathcal{K}[\![e_1]\!]\ (\lambda x_1.\ \mathcal{K}[\![e_2]\!]\ (\lambda x_2.\ x_1\ \langle x_2, \kappa\rangle)) \\
 & \ldots &
\end{array}
$$

Figure 2. CPS conversion

In the next section, we discuss how this transformation preserves types.

5.1 Type preservation

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

In order to prove this 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 5.2. *($\lambda_{\to}$–λ_K type correspondence) If* $\bullet \vdash e : \tau$*, then* $\bullet \vdash_{\mathcal{K}} \lambda c.\ \mathcal{K}[\![e]\!]\ c : (\mathcal{K}_{type}[\![\tau]\!] \to 0) \to 0$.

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.

5.2 Implementation

Encoding type correspondence At first approximation, by applying the Curry-Howard isomorphism, the type correspondence prop-

erty of the CPS transform (Theorem 5.2) might be reflected in the type of its implementation in this way:

```
cps :: Exp t -> (ValK K_type[[t]] -> ExpK) -> ExpK
```

where `Exp` represent $\lambda_{\rightarrow}$ terms, and `ValK` and `ExpK` represent $\lambda_{\mathcal{K}}$ values and expressions, respectively. 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 [Harper and Morrisett, 1995, Trifonov et al., 2000]. In the absence of this feature, we encode the relation between a type and its converted form using yet another GADT:

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

A term of type `CpsForm t cps_t` represents a proof that `cps_t` $= \mathcal{K}_{\text{type}}[\![\mathtt{t}]\!]$, and the type of `cps` would rather be:

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

Constructing and examining such witnesses is indeed cumbersome. In particular, it requires that we prove separately that $\mathcal{K}_{\text{type}}[\![-]\!]$ is indeed a function:

```
cpsUnique :: CpsForm t cps_t -> CpsForm t cps_t'
               -> Equal cps_t cps_t'

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

This fact is needed, for example, when converting a function application, where we need to convince the type checker that the converted function and its actual argument agree in types. In the remainder of this paper, we will freely use functions like $\mathcal{K}_{\text{type}}[\![-]\!]$ in type expressions, with the implied meaning of using explicit witnesses in the actual implementation.

Concrete representation When saying that it would be represented in this way:

```
data Exp t where
  Let :: Exp t1 -> (Exp t1 -> Exp t2) -> Exp t2
  ...
```

we overlooked a number of details of the concrete representation. In practice, we would rather use a type like this one:

```
data ExpF (α t) where
  Let :: α t1 -> (α t1 -> α t2) -> ExpF (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` would be represented as a Haskell term of type $\forall\alpha.$ `Exp` α `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). In the case at hand, we obtain `cps` by applying `cata` with β `t` instantiated at the type:

```
type CPS α t = (ValK α K_type[[t]] -> ExpK α) -> ExpK α
```

(*type env*)	$\Gamma ::= \bullet \mid \tau, \Gamma$
(*values*)	$v ::= \lambda\, e \mid j \mid \ldots$
(*exps*)	$e ::= \mathsf{let}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ v_1\ p\ v_2\ \mathsf{in}\ e \mid \ldots$
(*indices*)	$j ::= i_0 \mid i_1 \mid \ldots$

Figure 3. The CPS language in de Bruijn form ($\lambda^b_{\mathcal{K}}$)

$$
\begin{array}{l}
\mathsf{let}\ a = 1.8 \\
\quad b = 32 \\
\quad c = 24 \\
\quad c2b = \lambda arg\ .\ \mathsf{let}\ x = arg.0 \\
\qquad\qquad\qquad\quad k = arg.1 \\
\qquad\qquad\qquad\quad v_0 = a \cdot x \\
\qquad\qquad\qquad\quad v_1 = v_0 + b \\
\qquad\qquad\qquad \mathsf{in}\ k\ v_1 \\
\mathsf{in}\ c2b\ \langle c, \lambda v\ .\ \mathsf{halt}\ v\rangle
\end{array}
\quad\Longrightarrow\quad
\begin{array}{l}
\mathsf{let}\ 1.8 \\
\quad 32 \\
\quad 24 \\
\quad \lambda\ \mathsf{let}\ i_0.0 \\
\qquad\qquad i_1.1 \\
\qquad\qquad i_5 \cdot i_1 \\
\qquad\qquad i_0 + i_5 \\
\qquad\quad \mathsf{in}\ i_2\ i_0 \\
\mathsf{in}\ i_0\ \langle i_1, \lambda\ \mathsf{halt}\ i_0\rangle
\end{array}
$$

Figure 4. Example program converted to de Bruijn indices

Syntactic classes Ideally, we would like to define `ValK` and `ExpK` as two separate, mutually recursive types. However our fixed point operator (`Rec`) can only be applied to a single type, so instead we use the same type for the two syntactic categories:

```
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)))
    -- expressions
    Klet  :: a (V t) -> (a (V t) -> a Z)
                      -> ExpKF (a Z)
    ...

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

As can be seen, 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.

Danvy and Filinski's CPS transform Our compiler actually implements Danvy and Filinski's one-pass CPS conversion [Danvy and Filinski, 1992], where administrative redexes are reduced on-the-fly. As shown in [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 (*cps-obj*) instead of a meta-level one (*cps-meta*):

```
type CPS α t =
 ((ValK α K_type[[t]] -> ExpK α) -> ExpK α,    – cps-meta
  ValK α (K_type[[t]] -> Z) -> ExpK α)         – cps-obj
```

6. Conversion to de Bruijn indices

This section addresses the task of converting HOAS to de Bruijn form. We refrained from the temptation to let Template Haskell do the work, as we did for type checking, since the performance argument does not really hold here. The exercise led us to realize that the lack of explicit type contexts in the higher-order representation rules out fully static checking of this conversion's type safety, as discussed below.

The differences between the original CPS language ($\lambda_{\mathcal{K}}$) and its de Bruijn variant ($\lambda^b_{\mathcal{K}}$) are summarized on Fig. 3. A variable x is represented by an index i_n: the index i_0 refers to the nearest binder (irrespective of whether it is introduced by λ or let), i_1 refers to the second nearest binder, etc. The syntactic constructs for let and λ do not mention variable names. Type environments take the form $\tau_0, \ldots, \tau_{n-1}, \bullet$, meaning that each index in scope i_k has type τ_k.

The effect of translating $\lambda_{\mathcal{K}}$ into $\lambda^b_{\mathcal{K}}$ is illustrated on Fig. 4, where it is applied the CPS-converted example program. The two programs are line-by-line equivalent.

Implementation The key point in converting to de Bruijn indices is that a index simply reflects the difference between the type context Γ where the variable is defined, and the context Γ' where the variable is used. As in Section 5.2, the conversion operates over a HOAS representation, so the implementation is obtained by applying the appropriate catamorphism (here, $\lambda_{\mathcal{K}}$'s). This time, the result of the computation has type:

$$\beta\ \texttt{t} = \forall \texttt{ts.}\ \texttt{EnvRep ts -> ExpKb ts t}$$

where `ts` is a Haskell type that stands for Γ, `EnvRep ts` reifies this context as a Haskell term, and `ExpKb ts` represents a well-typed $\lambda^b_{\mathcal{K}}$ expression in type context Γ.

By parameterizing the resulting term by the type context, we are able to compare the type context where the binder is introduced and the context where the variable occurs. The difference in length between the two contexts tells us the de Bruijn index to put in place of the variable:

```
mkIndex :: Ctxrep (t, ts) -> CtxRep ts'
             -> ∃i. Index ts' i t
```

where `ts` and `ts'` are the Haskell types that stand for Γ and Γ'. For `mkIndex` to succeed, Γ' must actually be an extension of Γ, in the sense that new types may have been added to the initial context Γ to form the new context. Although it is indeed expected to always be the case, the types we use do not statically guarantee it; in consequence, `mkIndex` needs to compare the common part of `ts'` and `(t, ts)` to prove that they match.

Limitations This explicit comparison of type contexts is far from satisfactory. This indeed is testing, not verification – but can we do better?

In HOAS, the body of the let is represented by a function of type α `s -> `α `t`. Given this type, the relationship between the initial static context and the context at the point where a variable occurs simply cannot be expressed. The best we can do is to have `ts` appear in α, thus in effect propagating `ts` unchanged. This does not express how `ts` gets *extended* as binding forms are traversed.

Thus, to explicitly capture context extensions in HOAS would require deep changes to the representation. The conversion to de Bruijn being an artifact introduced as a consequence of our subjective choice of encoding, we found little motivation to look deeper. We leave it to future work to investigate a HOAS representation that would uncover a closer relationship to a typed de Bruijn representation as used here.

7. Closure conversion

The implementation of closure conversion turns out to be substantially more involved than CPS conversion. One reason for this is that the type of the output term depends on the result of a program analysis (computing the free variables) – this is in contrast to CPS conversion which is a direct, purely syntax-directed transformation. Another source of complexity is the use of de Bruijn indices, as it forces us to spell out the type safety proof in greater detail and to structure the algorithm in a somewhat peculiar manner. In counterpart, the resulting implementation has far improved precision w.r.t

$$\begin{array}{ll}
(\textit{types}) & \tau ::= \mathsf{closure}\ \tau \mid \ldots \\
(\textit{values}) & v ::= \mathsf{closure}\ e_f\ e_{env} \mid \ldots \\
(\textit{exps}) & e ::= \mathsf{let\ open}\ e_1\ \mathsf{in}\ e_2 \mid \ldots
\end{array}$$

Typing rules:

$$\frac{\bullet \vdash_C e_f : (\tau \times \tau_{env}) \to 0 \quad \Gamma \vdash_C e_{env} : \tau_{env}}{\Gamma \vdash_C \mathsf{closure}\ e_f\ e_{env} : \mathsf{closure}\ \tau}$$

$$\frac{\Gamma \vdash_C e_1 : \mathsf{closure}\ \tau \quad \tau_{env}, (\tau_1 \times \tau_{env}) \to \tau_2, \Gamma \vdash_C e_2}{\Gamma \vdash_C \mathsf{let\ open}\ e_1\ \mathsf{in}\ e_2}$$

Figure 5. The target language of closure conversion ($\lambda^b_{\mathcal{C}}$)

$$\begin{array}{rcl}
\mathcal{C}_b[\![i]\!]m & = & m\ i \\
\mathcal{C}_b[\![\mathsf{let}\ e_1\ \mathsf{in}\ e_2]\!]m & = & \mathsf{let}\ \mathcal{C}_b[\![e_1]\!]m \\
 & & \mathsf{in}\ \mathcal{C}_b[\![e_2]\!](i_0 : \mathit{map\ shift}\ m) \\
\mathcal{C}_b[\![\lambda\ e]\!]m & = & \mathsf{closure}\ (\lambda\ e_{body})\ e_{env} \\
\multicolumn{3}{r}{\text{where } (m', [j_0, \ldots, j_{n-1}]) = \mathit{mkEnv}\ (\mathit{tail}\ (\mathit{fvs}\ e))} \\
\multicolumn{3}{r}{e_{body} = \mathsf{let}\ i_0.0 \quad \textit{(original argument)}} \\
\multicolumn{3}{r}{\qquad\qquad i_1.1 \quad \textit{(environment)}} \\
\multicolumn{3}{r}{\qquad \mathsf{in}\ \mathcal{C}_b[\![e]\!](i_1 : \mathit{map}\ (\lambda j\ .\ i_0.j)\ m')} \\
\multicolumn{3}{r}{e_{env} = \langle m\ j_0, \ldots, m\ j_{n-1} \rangle}
\end{array}$$

$$\begin{array}{rcl}
\mathit{mkEnv}\ []\ j & = & ([], []) \\
\mathit{mkEnv}\ (False, b_1, \ldots, b_p)\ j & = & ((\bot : m), [j_0, \ldots, j_{n-1}]) \\
\mathit{mkEnv}\ (True, b_1, \ldots, b_p)\ j & = & ((n : m), [j_0, \ldots, j_{n-1}, j]) \\
\multicolumn{3}{r}{\text{where } (m, [j_0, \ldots, j_{n-1}]) = \mathit{mkEnv}\ [b_1, \ldots, b_p]\ (j+1)}
\end{array}$$

$$\mathit{fvs}\ e = [b_0, \ldots, b_{n-1} \mid b_i = True \text{ if } i_i \text{ appears in } e;\ False \text{ otherwise}]$$

$$\begin{array}{rcl}
\mathit{shift}\ i_n & = & i_{n+1} \\
\mathit{shift}\ i_n.k & = & i_{n+1}.k
\end{array}$$

Figure 6. Closure conversion over $\lambda^b_{\mathcal{K}}$

$$\begin{array}{ll}
\mathsf{let}\ a = 2 & \mathsf{let}\ 2 \\
\quad b = 4 & \quad 4 \\
\quad c = 7 & \quad 7 \\
\quad d = 8 & \quad 8 \\
\mathsf{in}\ \lambda x\ .\ a \cdot x + c & \mathsf{in}\ \lambda\ i_4 \cdot i_0 + i_2 \\
 & \\
 & \Downarrow \mathcal{C}_b[\![-]\!]- \\
 & \\
\mathsf{let}\ a = 2 & \mathsf{let}\ 2\ 4\ 7\ 8 \\
\quad b = 4 & \mathsf{in\ closure}\ (\lambda\ \mathsf{let}\ i_0.0 \\
\quad c = 7 & \qquad\qquad i_1.1 \\
\quad d = 8 & \qquad\qquad \mathsf{in}\ i_0.0 \cdot i_1 + i_0.1) \\
\mathsf{in\ closure}\ (\lambda arg\ .\ \mathsf{let}\ x = arg.0 & \qquad \langle i_3, i_1 \rangle \\
\qquad\qquad env = arg.1 & \\
\qquad\qquad a = env.0 & \\
\qquad\qquad c = env.1 & \\
\qquad \mathsf{in}\ a \cdot x + c) & \\
\qquad \langle a, c \rangle &
\end{array}$$

Figure 7. Example of closure conversion with variable names (left) and de Bruijn indices (right)

binders. It neatly captures not only the correspondence between the type of the input and output terms, but also the one that holds between a closure's environment and its functional part as they are constructed, a central point for proving type safety.

We proceed as follows: We first argue in favour of de Bruijn indices for the purpose of closure conversion and hosting, and then present the algorithm and illustrate its workings by applying it on a simple program, before going through the salient features of its Haskell implementation.

Justifications Although our initial intent was to use HOAS throughout all compilation phases, we soon realized that doing so for closure conversion and hoisting had a price we were not willing to pay:

1. The fact that HOAS does not represent variables explicitly has the unfortunate consequence that variables cannot be identified: given two variables a and b, we cannot (directly) determine whether the two variables are actually the same – which is essential for closure conversion. To recover this ability, one needs to somehow "inject" identity into variables, for example by annotating binders with some sort of names or indices. This approach tends to negate the chief advantages of HOAS, namely its conciseness and elegance. One would argue that such an "augmented" representation makes HOAS degenerate into something actually more complex than de Bruijn indices.
2. The fact that HOAS handles type environments implicitly precludes explicit constraints on type contexts, such as terms being closed. However, the hoisting transformation actually relies on the fact that functions inside closures are closed.

This is not to say that closure conversion over HOAS is impossible, but doing so would inevitably involve non-conventional extensions to the basic representation. So we settled for de Bruijn indices, conceding a little ground in elegance and conciseness: The result of this re-structuring is the algorithm shown in Fig. 6, carefully formulated to make it typeable.

The algorithm The target language of closure conversion ($\lambda^b_{\mathcal{C}}$, shown in Fig. 5) extends $\lambda^b_{\mathcal{K}}$ with syntactic forms for constructing and opening closures. We will refer to typing judgements $\Gamma \vdash_C e$ over $\lambda^b_{\mathcal{C}}$ expressions and $\Gamma \vdash_C v : \tau$ over $\lambda^b_{\mathcal{C}}$ values; these judgements enforce a type system identical to that of $\lambda^b_{\mathcal{K}}$, with the addition of the typing rules for closures (as shown).

The closure conversion algorithm is defined in Fig. 6, where $\mathcal{C}_b[\![e]\!]m$ denotes the closure-converted form of a $\lambda^b_{\mathcal{K}}$ expression e given local variables map m. The map m, for a source term with n variables in scope, has form $[e_0, \ldots, e_{n-1}]$, where e_k gives the local binding in the target program for source variable i_k. In general, e_k will be either a de Bruijn index (when i_k is a local variable of the function being converted) or a projection of the environment (when i_k is a free variable.) The definition of $\mathcal{C}_b[\![-]\!]-$ refers to auxiliary functions *mkEnv* and *fvs* that are used for constructing the map m when forming closures.

To illustrate how this works, we'll go through the closure conversion of a simple program. For simplicity, the example is written in direct style rather than CPS; there is non loss of generality, as there is basically no interaction between our CPS and closure conversion phases. The example is shown in Fig. 7. It is shown with variable names on the left, so as to make it easier to follow. The first step computes the free variables:

$$\mathit{fvs}\ (i_4 \cdot i_0 + i_2) = [\mathsf{True}, \mathsf{False}, \mathsf{True}, \mathsf{False}, \mathsf{True}]$$

meaning that i_0 appears in the term, and so do i_2 and i_4, but not i_1 or i_3. Next is the construction of the environment and the corresponding local variables map, which is handled by *mkEnv*.

$$\begin{array}{rcl} \mathcal{C}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int} \\ \mathcal{C}_{\text{type}}[\![\tau \to 0]\!] & = & \mathsf{closure}\ \mathcal{C}_{\text{type}}[\![\tau]\!] \\ \\ \mathcal{C}_{\text{env}}[\![\bullet]\!] & = & \bullet \\ \mathcal{C}_{\text{env}}[\![\tau, \Gamma]\!] & = & \mathcal{C}_{\text{type}}[\![\tau]\!], \mathcal{C}_{\text{env}}[\![\Gamma]\!] \end{array}$$

Figure 8. Correspondence between types (en environments) in $\lambda^b_{\mathcal{K}}$ and $\lambda^b_{\mathcal{C}}$.

We have:

$$\begin{array}{rl} & (m', [j_0, \ldots, j_{n-1}])\ 0 \\ = & \mathit{mkEnv}\ (\mathit{tail}\ (\mathit{fvs}\ (i_4 \cdot i_0 + i_2)))\ 0 \\ = & \mathit{mkEnv}\ (\mathit{tail}\ [\mathsf{True}, \mathsf{False}, \mathsf{True}, \mathsf{False}, \mathsf{True}])\ 0 \\ = & \mathit{mkEnv}\ [\mathsf{False}, \mathsf{True}, \mathsf{False}, \mathsf{True}]\ 0 \\ = & ([\bot, 1, \bot, 0], [i_3, i_1]) \end{array}$$

The second component ($[i_3, i_1]$), simply enumerates the source indices to be put in the environment. The first one, m', maps variables in scope in the function's body (except the function's original argument, i_0) to corresponding projections of the environment. From this m', $\mathcal{C}_b[\![-]\!]-$ constructs a map in which to interpret the function's body:

$$(i_1 : \mathit{map}\ (\lambda j\ .\ i_0.j)\ m') = [i_1, \bot, i_0.1, \bot, i_0.0]$$

Finally, the function's body can be converted:

$$\mathcal{C}_b[\![i_4 \cdot i_0 + i_2]\!][i_1, \bot, i_0.1, \bot, i_0.0] = i_0.0 \cdot i_1 + i_0.1$$

7.1 Type preservation

As for the CPS transform, type preservation states that closure conversion takes well typed programs to well typed programs:

THEOREM 7.1. *(CC type preservation) For any $\lambda^b_{\mathcal{K}}$ expression e, if $\bullet \vdash_K e$, then $\bullet \vdash_C \mathcal{C}[\![e]\!]$.*

In reality, there is a close correspondence between types in L_S and those in L_C. That correspondence between types (and type environments) is captured by the relation $\mathcal{C}_{\text{type}}[\![-]\!]$ (and $\mathcal{C}_{\text{env}}[\![-]\!]$) defined in Fig. 8.

We can now be more precise about the type of the converted term, and generalize the statement to open terms:

THEOREM 7.2. *(CC type correspondence)*

1. *For any $\lambda^b_{\mathcal{K}}$ expression e, if $\Gamma \vdash_K e$, then $\Gamma \vdash_C \mathcal{C}_{env}[\![\Gamma]\!]\mathcal{C}[\![e]\!]$, and*
2. *for any $\lambda^b_{\mathcal{K}}$ value v, if $\Gamma \vdash_K v : \tau$, then $\mathcal{C}_{env}[\![\Gamma]\!] \vdash_C \mathcal{C}[\![v]\!] : \mathcal{C}_{type}[\![\tau]\!]$.*

The above theorem captures the key invariant that guarantees type preservation: an index j of type τ in the source program is mapped to an index j' of type $\mathcal{C}_{\text{type}}[\![\tau]\!]$ in the target program. In particular, when constructing a closure, every variable referenced in the body of the closure is bound to a value (extracted from the environment) of the expected type.

7.2 Implementation

This final section provides a brief outline of the way in which the technical details of closure conversion over de Bruijn terms translate into Haskell types. After reviewing the program representation, we define a notion of type-preserving maps over type contexts, that serves as building block with which we construct the type of $\mathcal{C}_b[\![-]\!]-$ and, in turn, that of idfvs and *mkEnv*.

Program representation As mentioned above, a first-order representation with explicit type contexts allows us to express the fact

that functions inside closure are closed. In addition, GADT's existential types can naturally be used to hide the type of the environment when forming a closure.

```
data ExpCb ts t where
  CBVclosure :: ExpCb () ((s, env) -> Z)
             -> ExpCb ts env
             -> ExpCb ts (V (Closure s))
  CBopen :: ExpCB ts (Closure s)
             -> (forall env.
                 ExpCb (((s, env) -> Z, env), ts) Z)
             -> ExpCb ts Z
  ...
```

Type-preserving maps Conceptually, a type-preserving map, of type `MapT ts c`, associates each index of type `Index ts i t` with a value of type `c t`.

```
data MapT ts c where
  M0 :: MapT () c
  Ms :: c t -> MapT ts c -> MapT (t, ts) c
```

For example, a type-safe evaluator over de Bruijn expressions might be given the type:

```
eval :: MapT ts Value -> ExpS ts t -> Value t
```

where the evaluation environment (`MapT ts Value`) maps each variable in scope (of type τ) to a value of the corresponding type (of type `Value` τ.) The type `MapT` supports the usual functions over associative lists:

```
lookupT  :: MapT ts c -> Index ts i t -> c t
updateT  :: MapT ts c -> Index ts i t -> c t
             -> MapT ts c
```

Type correspondence Again, we encode the correspondence between the type τ (for which `t` stands) and its converted form $\mathcal{C}_{\text{type}}[\![\tau]\!]$ (for which `cc_t` stands) using a GADT:

```
data CC_type t cc_t where
  CCint :: CC_type Int Int
  CCfun :: CC_type s cc_s -> CC_type t cc_t
             -> CC_type (s -> t) (Closure cc_s cc_t)
```

and encode correspondence on type environments ($\mathcal{C}_{\text{env}}[\![-]\!]$) similarly.

The function $\mathcal{C}_b[\![-]\!]m$ receives a source term (in context `ts`) and a local variables map (mapping `ts` indices to indices in some target context `ts'`), and produces an expression (in the target context `ts'`). Its type is the following:

```
cc :: ExpS ts t
  -> MapT C_env[[ts]] (SomeIndex ts')
  -> ExpC ts' C_type[[t]]
```

where `SomeIndex ts t` is a type that abstracts away from the numeric value of the target index:

```
type SomeIndex ts t = ∃i. Index ts i t
```

The map's domain is chosen to be $\mathcal{C}_{\text{env}}[\![\texttt{ts}]\!]$ rather than `ts` in order to get a type-safe mapping to variables in the closure converted program. It dos not cause a technical problem, as `ts` and $\mathcal{C}_{\text{env}}[\![\texttt{ts}]\!]$ are indeed isomorphic.

Free variables The `fvs` function, given an expression, indicates whether each index in scope appears in the expression. Its implementation produces its result in the type `MapT`:

```
fvs :: ExpS ts t -> MapT ts BoolT
```

where `BoolT` is a wrapper for the type `Bool` that has an extra type argument `t` that is simply ignored:

$$
\begin{array}{lll}
(\textit{programs})\ p & ::= & \mathsf{letrec}\ e_0 \\
 & & \vdots \\
 & & e_{n-1} \\
 & & \mathsf{in}\ e \\
(\textit{indices})\ j & ::= & \ldots \mid \ell_n
\end{array}
$$

Figure 9. Target language of the hoisting transformation ($\lambda^b_{\mathcal{H}}$)

$$
\begin{array}{l}
\mathit{collect}\ \ell_m\ j \quad = \quad ([], j) \\
\\
\mathit{collect}\ \ell_m\ (\lambda\ e) \quad = \quad ([\lambda\ e', e_{m+1}, \ldots, e_n], \ell_m) \\
\quad \text{where } ([e_{m+1}, \ldots, e_n], e') = \mathit{collect}\ \ell_{m+1}\ e \\
\\
\mathit{collect}\ \ell_m\ (\mathsf{let}\ e_1\ \mathsf{in}\ e_2) \quad = \quad ([e_m, \ldots, e_n, e_{n+1}, \ldots, e_p], \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ e_1'\ \mathsf{in}\ e_2') \\
\quad \text{where } ([e_m, \ldots, e_n], e_1') = \mathit{collect}\ \ell_m\ e_1 \\
\qquad\quad ([e_{n+1}, \ldots, e_p], e_2') = \mathit{collect}\ \ell_{n+1}\ e_2 \\
\ldots \\
\\
\mathit{hoist}\ e = \mathsf{letrec}\ e_0 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad e_{n-1} \\
\qquad\qquad\qquad \mathsf{in}\ e' \\
\quad \text{where } ([e_0, \ldots, e_{n-1}], e') = \mathit{collect}\ \ell_0\ e
\end{array}
$$

Figure 10. Hoisting transformation (transforms $\lambda^b_{\mathcal{C}}$ into $\lambda^b_{\mathcal{H}}$)

```
data BoolT t = BoolT Bool
```

Closure environment construction The function *mkEnv* in essence consumes the list of free variables and produces two results: (1) a local variables map, mapping each index in scope to a projection of the environment, and (2) a list of indices to be packed in the environment. There is of course a direct connection between the two: the local variables map assumes a target context formed out of the environment being constructed. We can readily express this in types as follows:

```
mkEnv :: MapT ts BoolT
      -> ∃env. (MapT C_env[[ts]] (SomeIndex env),
                MapT env C_env[[ts]])
```

8. Hoisting

The techniques developed for closure conversion are in large part applicable to hoisting as well, and the implementation is structured in a similar manner, but it is indeed simpler, as one would expect.

The target language is shown in Fig. 9. It extends $\lambda^b_{\mathcal{K}}$ with a syntactic category of *programs*, providing a top-level letrec construct. The letrec construct introduces a number of indices $\ell_0, \ldots, \ell_{n-1}$; the scope of all those indices spans the body of all the letrec-bindings ($e_0, \ldots, e_{n-1}$) plus the program body (e). The binders introduced by letrec ($\ell_0, \ldots, \ell_{n-1}$) form a set of indices distinct from those introduced by λ or let (that is, $i_0, i_1 \ldots$).

The hoisting transformation is shown in Fig. 10. The auxiliary function *collect*, as its name implies, collects the λ-abstractions contained in a source term. Its first argument ℓ_m indicates the smallest unassigned index (that is, the smallest value of m for which the binders $\ell_0 \ldots \ell_{m-1}$ are already assigned to λ-abstractions,

but ℓ_m is not.) The second argument gives the source term to convert. The result of $collect\ \ell_m\ e$ is a pair consisting of:

1. a list of λ-abstractions $e_m \ldots e_n$, where each e_k is assigned the binder ℓ_k, and each sub-term of e_k that is λ-abstraction is replaced by its assigned binder, and
2. the converted form of e, that is, e with each λ-abstraction subterm replaced by its assigned binder.

8.1 Implementation

We first describe a program representation for $\lambda^b_{\mathcal{H}}$, and then outline the main features of the implementation of $collect$ and $hoist$, which mainly concern the way the types of the expressions $e_0 \ldots e_{n-1}$ is constructed as $collect$ proceeds.

Program representation The letrec construct of $\lambda^b_{\mathcal{H}}$ introduces a number of bindings by listing the expressions $(e_0 \ldots e_{n-1})$ associated with each respective binder $(\ell_0 \ldots \ell_{n-1})$; the bundle of expressions $(\ell_0 \ldots \ell_{n-1})$ can be represented with the usual type for tuple formation $(\langle e_0, \ldots, e_{n-1} \rangle)$:

```
data Tuple ts t where
  B0 :: Tuple ts ()
  Bs :: ExpH ts s -> Tuple ts t
          -> Tuple ts (s, t)
```

where the first type parameter, `ts`, reflects the De Bruijn context of every expression in the tuple, and the second type parameter, `t`, reflects the type of the tuple itself. To get a bundle of mutually recursive terms, we take `ts` $=$ `t`:

```
data Program t where
  Letrec :: Tuple ts ts -> ExpH ts t -> Program t
```

Collecting λ-abstractions The parameter ℓ_m to the function $collect$ reflects the number of binders that have already been assigned λ-abstractions: when $collect$ meets a λ-abstraction, it readily assigns it to ℓ_m, knowing that it's the smallest unused index. Each time a λ-abstraction is assigned to a binder, the bundle of terms to be put in the letrec grows by one – and we have to track the type of the bundle of functions as it grows when recursive calls to $collect$ are made.

9. Related work

Closure conversion is a well-studied problem, both from a performance point of view [Shao and Appel, 1994], as well as its interaction with types [Minimide et al., 1996, Morrisett et al., 1998]. For obvious reasons we use a fairly naive algorithm, and since our source language is simply typed, we are not affected by the potential difficulties linked to closure conversion of polymorphic code.

There has been a lot of work on typed intermediate languages, beginning with the TIL [Tarditi et al., 1996] and FLINT [Shao and Appel, 1995, Shao, 1997b] work, originally motivated by the optimizations opportunities offered by the extra type information. [Necula, 1997] 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 [Morrisett et al., 1999].

In [Shao et al., 2002], 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 [Pasalic, 2004], Emir Pašalić 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 de Bruijn indices and relies on type equality proofs in Haskell.

In [Chen and Xi, 2003], 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 the explicit manipulation of proof terms, but they have made other design trade-offs: their term representation is first order using de Bruijn indices, and their implementation language is more experimental. In a similar vein, Linger and Sheard [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. In [Guillemette and Monnier, 2006], we showed the details of the CPS phase of our compiler, where the distinguishing feature is the use of a term representation based on HOAS and in [Guillemette and Monnier, 2007] we showed the details of our closure conversion phase where we switched to de Bruijn indices.

In [Leroy, 2006], 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. This was pushed further to the front-end in [Blazy et al., 2006]. The goal of that project is a lot more ambitious since they prove that the dynamic semantics of the output programs are faithful to their source code, whereas we only verify type preservation. In [Chlipala, 2007], Chlipala presents a similar work which is even closer to ours since his source language is also the simply typed λ-calculus. There as well, the programming language used is Coq and he focuses on dynamic semantics, although he also does use a typed representation similar to ours (using de Bruijn indices).

In [Fegaras and Sheard, 1996], Fegaras and Sheard show how to handle higher-order abstract syntax, and in [Washburn and Weirich, 2003], 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. Recently, Brigitte Pientka [Pientka, 2006] presented a new language where programming in HOAS is made simpler by providing special support for it.

GADTs were introduced many times under many different names [Xi et al., 2003, Cheney and Hinze, 2003, Sheard and Pašalić, 2004]. Their interaction with type classes is a known problem in GHC and a possible solution was proposed in [Sulzmann et al., 2007] and is in the process of being implemented.

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. doi: http://doi.acm.org/10.1145/777388.777392.

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.

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, aug 2006.

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

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.

Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*, pages 130–141, January 1995.

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. doi: http://doi.acm.org/10.1145/1111037.1111042.

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

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

Yasuhiko Minimide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, January 1996.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Symposium on Principles of Programming Languages*, pages 85–97, January 1998.

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.

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. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54010.

Brigitte Pientka. Functional programming with higher-order abstract syntax and explicit substitutions. In *Programming Languages meets Program Verification*, volume 174(7) of *Electronic Notes in Theoretical Computer Science*, pages 41–60, aug 2006.

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

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

Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 150–161, Orlando, FL, June 1994.

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. doi: http://doi.acm.org/10.1145/581690.581691.

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

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*, jan 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.

Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *International Conference on Functional Programming*, pages 82–93. ACM Press, September 2000.

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.