A Type-Preserving Closure Conversion 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. Also it guarantees that any property that was enforced by the source-level type-system is holds also for the generated code. Recently, several people have tried to push this effort a bit further by *verifying* formally that the compiler indeed preserves typing. This is usually done with proof assistants or experimental languages.

Instead, we decided to use Haskell, to see how far we can go with a more mainstream system, supported by robust compilers and plentiful libraries. This article presents one part of our type preserving compiler, namely the closure conversion and its associated hoisting phase, where we use GADTs to let Haskell's type checker verify the we obey the object language's typing rules and that we correctly preserve types from one phase to the other.

This should be both a good showcase as well as a good stress test for GADTs, so we also discuss our experience, as well as some trade-offs in the choice of representation, namely between higher-order abstract syntax (HOAS) and a first order representation (i.e. de Bruijn indices) and justify our choice of a de Bruijn representation. We incidentally present a type preserving conversion from HOAS (used in earlier phases of the compiler [4]) to a de Bruijn representation.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

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 [6, 22, 15], sanity checks to help catch compiler errors, and more recently to help construct proofs that the generated code verifies some properties [11, 5]. 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:

- 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. [1]) 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 strenth 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 [7], 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

Source language (L_S)

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

Target language (L_C)

$$
\begin{array}{lll}
\textit{(types)} & \tau & ::= \ldots \mid \mathsf{closure}\ \tau_1\ \tau_2 \mid \tau_1 \times \cdots \times \tau_n \\
\textit{(exps)} & e & ::= \ldots \mid \mathsf{closure}\ e_f\ e_{env} \\
 & & \quad \mid \mathsf{let}\ (x_f, x_{env})\ =\ \mathsf{open}\ e_1\ \mathsf{in}\ e_2 \\
 & & \quad \mid \langle e_1, \ldots, e_n \rangle \mid e.i
\end{array}
$$

Typing rules (L_C)

$$
\frac{\bullet \vdash e_f : (\tau_1 \times \tau_{env}) \to \tau_2 \quad \Gamma \vdash e_{env} : \tau_{env}}{\Gamma \vdash \mathsf{closure}\ e_f\ e_{env} : \mathsf{closure}\ \tau_1\ \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{closure}\ \tau_1\ \tau_2 \qquad \Gamma, x_f : (\tau_1 \times \tau_{env}) \to \tau_2, x_{env} : \tau_{env} \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ (x_f, x_{env})\ =\ \mathsf{open}\ e_1\ \mathsf{in}\ e_2 : \tau_2}
$$

Figure 1. Source and target language

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 [4], we presented the CPS phase of our compiler, which used a higher order abstract syntax (HOAS) [14] 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 a type-preserving closure conversion written in Haskell with GHC extensions (mainly GADTs) and where the GHC type checker verifies the property of type-preservation.
- We similarly show a type-preserving function hoisting phase.
- We additionally show a type preserving conversion from strongly typed higher-order abstract syntax (HOAS) terms (following [23]) into strongly typed first order terms using de Bruijn indices.

The paper is structured as follows: We begin with background material on closure conversion, hoisting and generalized algebraic datatypes (GADTs) in Section 2. We present the details of closure conversion over a concrete program representation with De Bruijn indices, and outline the main features of our implementation in Haskell in Section 3; we do the same for hoisting in Section 4. Finally, we present a technique for converting a typeful higher-order abstract syntax (HOAS) representation into our first-order representation with de Bruijn indices in Section 5, before concluding with related and future work.

2. Background

In this section we describe what we mean by closure conversion, hoisting and type preservation, and briefly show the kind of typeful program representation using GADTs we use.

$$
\begin{array}{rcl}
\mathcal{C}[\![x]\!] & = & x \\
\mathcal{C}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ x = \mathcal{C}[\![e_1]\!]\ \mathsf{in}\ \mathcal{C}[\![e_2]\!] \\
\mathcal{C}[\![\lambda x\ .\ e]\!] & = & \mathsf{closure}\ (\lambda\langle x, x_{env}\rangle\ .\ e_{body})\ e_{env} \\
 & \text{where} & y_1, \ldots, y_n\ =\ FV(e) \\
 & & e_{body}\ =\ \mathsf{let}\ y_1 = x_{env}.1 \\
 & & \qquad\qquad \vdots \\
 & & \qquad\quad y_n = x_{env}.n \\
 & & \quad \mathsf{in}\ \mathcal{C}[\![e]\!] \\
 & & e_{env}\ =\ \langle y_1, \ldots, y_n \rangle \\
\mathcal{C}[\![e_1\ e_2]\!] & = & \mathsf{let}\ (x_f, x_{env}) = \mathsf{open}\ \mathcal{C}[\![e_1]\!] \\
 & & \mathsf{in}\ x_f\ \langle \mathcal{C}[\![e_2]\!], x_{env} \rangle \\
\mathcal{C}[\![n]\!] & = & n \\
\mathcal{C}[\![e_1\ p\ e_2]\!] & = & \mathcal{C}[\![e_1]\!]\ p\ \mathcal{C}[\![e_2]\!]
\end{array}
$$

Figure 2. Closure conversion

2.1 Closure conversion

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.

The source language (L_S) used here is a simply typed, call-by-value λ-calculus, with a non-recursive let-form and integers as a base type.[1] 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 L_S expressions, assuming standard definitions.

The target language (L_C) extends L_S with syntactic forms for constructing and opening closures, as well as n-tuples. The syntax[2] of the two languages, as well as the typing rules for closures, are shown in Fig. 1. We will refer to a typing judgement $\Gamma \vdash e : \tau$ over L_C expressions, which extends that for L_S with the typing rules for closures (as shown) and n-tuples.

The usual formulation of closure conversion is shown in Fig. 2. The result of closure conversion applied to a simple program is shown in Fig. 3. In the next section, we discuss how such transformation preserves types.

2.2 Type preservation

In its simplest form, type preservation states that closure conversion takes well typed programs to well typed programs:

THEOREM 2.1. *(CC type preservation) For any L_S expression e, if $\bullet \vdash e : \tau$, then $\bullet \vdash \mathcal{C}[\![e]\!] : \tau'$ for some L_C type τ'.*

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. 4.

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

THEOREM 2.2. *(CC type correspondence) For any L_S expression e, if $\Gamma \vdash e : \tau$, then $\mathcal{C}_{env}[\![\Gamma]\!] \vdash \mathcal{C}[\![e]\!] : \mathcal{C}_{type}[\![\tau]\!]$.*

[1] Although the languages used here are in direct style, our implementation actually preforms closure conversion over programs in CPS.

[2] To make programs easier to read, we freely use pattern matching (e.g. to introduce multiple bindings at once using tuple syntax), and use Haskell-style indentation to clarify program structure.

Source program (L_S):

$$\begin{array}{l}\mathsf{let}\ a = 2\\ \quad\ b = 5\\ \quad\ f = \lambda x\ .\ \lambda y\ .\ a \cdot x\ +\ b \cdot y\\ \mathsf{in}\ f\ 7\ 3\end{array}$$

Closure-converted program (L_C):

$$\begin{array}{l}\mathsf{let}\ a = 2\\ \quad b = 5\\ \quad f = \mathsf{closure}\ \lambda\langle x, env\rangle\ .\\ \qquad\qquad \mathsf{let}\ \langle a, b\rangle = env\\ \qquad\qquad \mathsf{in\ closure}\ \lambda\langle y, env\rangle\ .\\ \qquad\qquad\qquad\qquad \mathsf{let}\ \langle a, b, x\rangle = env\\ \qquad\qquad\qquad\qquad \mathsf{in}\ a \cdot x\ +\ b \cdot y\\ \qquad\qquad\qquad\quad \langle a, b, x\rangle\\ \qquad\quad \langle a, b\rangle\\ \quad \langle f_f, f_{env}\rangle = \mathsf{open}\ (\mathsf{let}\ \langle f_f, f_{env}\rangle = \mathsf{open}\ f\\ \qquad\qquad\qquad\qquad \mathsf{in}\ f_f\ \langle 7, f_{env}\rangle)\\ \mathsf{in}\ f_f\ \langle 3, f_{env}\rangle\end{array}$$

Figure 3. Example of closure conversion

$$\begin{array}{rcl}\mathcal{C}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int}\\ \mathcal{C}_{\text{type}}[\![\tau_1 \to \tau_2]\!] & = & \mathsf{closure}\ \mathcal{C}_{\text{type}}[\![\tau_1]\!]\ \mathcal{C}_{\text{type}}[\![\tau_2]\!]\\ \\ \mathcal{C}_{\text{env}}[\![\bullet]\!] & = & \bullet\\ \mathcal{C}_{\text{env}}[\![\Gamma, x{:}\tau]\!] & = & \mathcal{C}_{\text{env}}[\![\Gamma]\!], x{:}\mathcal{C}_{\text{type}}[\![\tau]\!]\end{array}$$

Figure 4. Correspondence between types (en environments) in L_S and L_C.

$$\begin{array}{l}\mathsf{let}\ \ell_0 = \lambda\langle y, env\rangle\ .\ \mathsf{let}\ \langle a, b, x\rangle = env\\ \qquad\qquad\qquad\qquad\ \mathsf{in}\ a \cdot x\ +\ b \cdot y\\ \quad \ell_1 = \lambda\langle x, env\rangle\ .\ \mathsf{let}\ \langle a, b\rangle = env\\ \qquad\qquad\qquad\qquad\ \mathsf{in\ closure}\ \ell_0\ \langle a, b, x\rangle\\ \quad a = 2\\ \quad b = 5\\ \quad f = \mathsf{closure}\ \ell_1\ \langle a, b\rangle\\ \quad \langle f_f, f_{env}\rangle = \mathsf{open}\ (\mathsf{let}\ \langle f_f, f_{env}\rangle = \mathsf{open}\ f\\ \qquad\qquad\qquad\qquad\ \mathsf{in}\ f_f\ \langle 7, f_{env}\rangle)\\ \mathsf{in}\ f_f\ \langle 3, f_{env}\rangle\end{array}$$

Figure 5. Program from Fig. 3 after hoisting.

The above theorem captures the key invariant that guarantees type preservation: a variable x of type τ in the source program is mapped to a variables of the same name x 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.

2.3 Hoisting

After closure conversion, λ-abstractions are closed and can be moved to the top level. (Note that the typing rules for closures actually *forces* functions to be closed: an open function inside a closure would simply not type check.)

For example, the result of the hoisting transformation applied to the program from Fig. 3 is shown in Fig. 5. In this example, the inner function ($\lambda\langle y, env\rangle\ldots.$) gets bound to ℓ_0, and the outer function ($\lambda\langle x, env\rangle\ldots.$) gets bound to ℓ_1.

It is easy to see that hoisting preserves types: a λ-abstraction is merely replaced by a variable occurrence of the same type.

2.4 Generalized algebraic datatypes

Generalized algebraic datatypes (GADTs) [24, 2] 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 the usual typing rule for function application:

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

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

```
data Exp where
  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`:

```
data Exp t where
  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. We will discuss a couple of ways of doing this in Section 3.1.

3. Closure conversion

In this section the core contribution of this paper is developed, the type-preserving closure conversion implemented with GADT's.

We begin with an overview of possible program representations, both first-order and higher-order, and justify our choice of de Bruijn indices. We then update our definition of closure conversion ($\mathcal{C}[\![-]\!]$) to work with de Bruijn indices, yielding a transformation ($\mathcal{C}_b[\![-]\!]-$) that is better amenable to typing. We give Haskell types for an implementation of $\mathcal{C}_b[\![-]\!]-$ and its auxiliary functions.

3.1 Choice of representation

There are at least a few ways in which the program representation from Section 2.4 can be extended with syntactic forms that introduce binders. We will illustrate two of them by showing how would be encoded the typing rule for let-expressions:

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

HOAS With higher-order abstract syntax, the typing rule for let 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 it does not require 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 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 `ExpB 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`.

To illustrate the two techniques, the following expression:

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

would be represented in HOAS as:

```
Elet (Enum 2) (λa ->
  Elet (Enum 3) (λb ->
    Eadd a b))
```

and with de Bruijn indices as:

```
Elet (Enum 2) (
  Elet (Enum 3) (
    Eadd (Is I0) I0))
```

Justifications 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. This ability is actually needed to perform closure conversion, as should become clear in Section 3. 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 – why not simply use de Bruijn indices, then?

Aside, as said earlier, 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, and de Bruijn's ability to enforce this is clearly an advantage.

Source language (L_S^b)

$$\begin{array}{ll}(\textit{indices}) & j ::= i_0 \mid i_1 \mid \ldots\\ (\textit{exps}) & e ::= j \mid \mathsf{let}\ e_1\ \mathsf{in}\ e_2 \mid \lambda\, e \mid \ldots\end{array}$$

Target language (L_C^b)

$$(\textit{exps}) \quad e ::= \ldots \mid \mathsf{let\ open}\ e_1\ \mathsf{in}\ e_2 \mid \ldots$$

Figure 6. Source and target language in de Bruijn form

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

$$\begin{array}{lcl}mkEnv\ []\ j & = & ([], [])\\ mkEnv\ (False : bs)\ j & = & ((\bot : m), [j_0, \ldots, j_{n-1}])\\ mkEnv\ (True : bs)\ j & = & ((n : m), [j_0, \ldots, j_{n-1}, j])\\ \multicolumn{3}{l}{\quad \text{where } (m, bs) = mkEnv\ [b_1, \ldots, b_{p-1}]}\end{array}$$

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

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

Figure 7. Closure conversion over L_S^b

In the face of these arguments in favour of a first-order encoding, we settled for de Bruijn indices for the task of closure conversion and hoisting, although we could probably have managed with HOAS, at the cost of having to extend the basic representation in non-conventional ways.

3.2 Closure conversion and de Bruijn indices

In this section we adjust the definition of $\mathcal{C}[\![-]\!]$ to work with de Bruijn indices. We first re-formulate the language definition of the source and target languages in de Bruijn form, then see precisely how closure conversion is affected, and show how it works by going through the details of converting a simple object program.

The de Bruijn form of L_S and L_C, that we call L_S^b and L_C^b, is shown in Fig. 6. The figure only shows the constructs relating to bindings, the others being left unchanged. 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, let, or open), i_1 refers to the second nearest binder, etc. The syntactic constructs for let, λ and open do not mention variable names. The form let open e_1 in e_2 introduces two bindings in e_2: i_0 is bound to the environment extracted from the closure e_1, and i_1 is bound to the function.

We now turn to closure conversion itself. The central part of closure conversion is that which converts functions to closures. In closure-converting the body of a λ-abstraction, one must arrange for (free) variable references to be turned into references to the corresponding variables stored in the environment. In the definition of $\mathcal{C}[\![-]\!]$, this is simply achieved by instantiating a number of

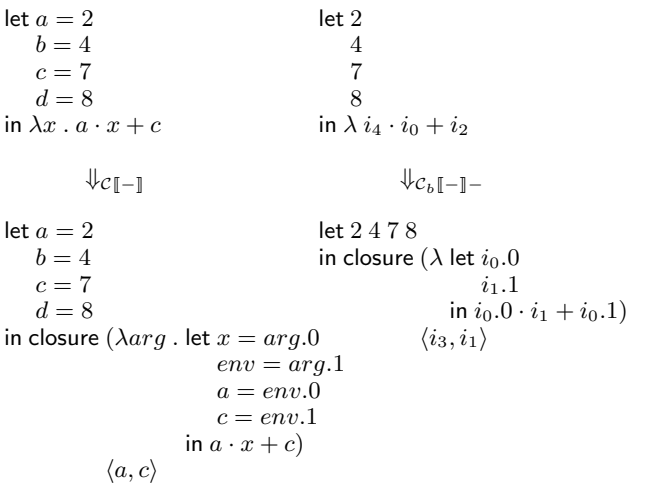

Figure 8. Example of closure conversion with variable names (left) and de Bruijn indices (right)

let-bindings with the *same names* as the original variables, each variable being bound to the corresponding value in the environment. For instance, in the example from Fig.3, the inner function $(\lambda y\ .\ a \cdot x\ +\ b \cdot y)$ gets converted into a closure whose body is syntactically identical to the original $(a \cdot x\ +\ b \cdot y)$, but whose variable refer to bindings that are *local* to the closure, each instantiated to a variable from the environment. Here, we wish to transpose this technique to our concrete representation with de Bruijn indices; but indeed, given that there are no variable names, we'll have to work a little harder.

Essentially, since we cannot rely on names, we'll have to carry around a map that gives the local binding in the converted program for each variable in scope in the source program. We denote $\mathcal{C}_b[\![e]\!]m$ the closure-converted form of source program e given local bindings m; the function $\mathcal{C}_b[\![-]\!]-$ in defined in Fig. 3.2. It refers to auxiliary functions $mkEnv$ and fvs that are used for constructing the map m when forming closures.

The local variables 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.)

To illustrate, consider the source program shown at the top of Fig. 8; the final result of the conversion is shown at the bottom. We now go through the steps involved in closure-converting this function.

The first step computes the free variables. Rather than producing a set, the fvs function produces a "bit-map", indicating whether each index in scope appears in the term. Taking the free variables of the function body, we have:

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

which reads, from left to right: i_0 appears in the term, i_1 does not, i_2 appears, and so on.

Next is the construction of the environment and the corresponding local variables map, which is handled by $mkEnv$. We have:

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

The first component, 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:

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

which reads, form left to right:

1. the source variable i_0 is mapped to local variable i_1,
2. the source variable i_1 is not mapped to any local variable, as indicated by $\bot$ (since the variable is in scope but does not appear in the source term, this is indeed what we want),
3. the source variable i_2 is mapped to $i_0.0$, the first projection of the environment,

and so on. The second component produced by $mkEnv$, namely $[j_0, \ldots, j_{n-1}]$, simply enumerates the source variables that appear in the function's body. Finally, the function's body can be converted:

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

What we've shown here is a mostly conventional formulation of closure conversion, only slightly contrived to facilitate typing; in the next section, we'll assign types to $\mathcal{C}_b[\![-]\!]-$, fvs and $mkEnv$.

3.3 Implementation

In this section we go through the implementation of closure conversion. We first define a notion of type-preserving maps over type contexts, which turns out to be a central concept; such a map associates to each variable in scope of type τ a piece of data whose type is indexed in τ. The primary interest for this structure is in typing the local variables map (m) of $\mathcal{C}_b[\![-]\!]m$: its key feature is that it maps variables in the source program to variables of the corresponding type in the converted program. Next, we see how to encore the relation $\mathcal{C}_{\text{type}}[\![-]\!]$ between types in the L^b_S and L^b_C. We then construct the type of $\mathcal{C}_b[\![-]\!]-$ and, in turn, that of fvs and $mkEnv$.

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 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'`); in types, this reads something like:

```
cc :: ExpS ts t
  -> MapT ...
  -> ExpC ts' C_type[[t]]
```

This way of writing $\mathcal{C}_{\text{type}}[\![-]\!]$ (c.f. Fig. 4) in type expressions is an abuse of notation: Haskell does not currently support type-level functions defined by case analysis. In the absence of this feature, we encode the relation between a type and its converted form using yet another 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)
```

A term of type `CC_type t cc_t` is a witness of the correspondence between the type τ (for which `t` stands) and its converted form $\mathcal{C}_{\text{type}}[\![\tau]\!]$ (for which `cc_t` stands). We define a type `CC_env` similarly to encode $\mathcal{C}_{\text{env}}[\![-]\!]$, and the type of `cc` is in fact:

```
cc :: ExpS ts t
   -> MapT ...
   -> ∃cc_t. (CC_type t cc_t, ExpC ts' cc_t)
```

Constructing and examining such witnesses is indeed cumbersome. It requires, for instance, a Haskell function encoding a proof that $\mathcal{C}_{\text{type}}[\![-]\!]$ (or $\mathcal{C}_{\text{env}}[\![-]\!]$) is indeed a function. In the remainder of this paper, we will freely use functions like $\mathcal{C}_{\text{type}}[\![-]\!]$ in type expressions, with the implied meaning of using explicit witnesses in the actual implementation.

Local variables map The map m passed to $\mathcal{C}_b[\![]\!]m$ takes each source index to an index of the converted type in the target context `ts'`:

m : `Index ts i t` $\Rightarrow$ `∃i'. Index ts' i'` $\mathcal{C}_{\text{type}}[\![\texttt{t}]\!]$

We'll have to make a few manipulations to make this map an instance of `MapT`. First, we define a type[3] that abstracts away from the numeric value of the target index (`i'`):

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

so that we have:

m : `Index ts i t` $\Rightarrow$ `SomeIndex ts'` $\mathcal{C}_{\text{type}}[\![\texttt{t}]\!]$

What keeps us from instantiating `MapT` is that the domain is indexed in `t`, while the image is indexed in $\mathcal{C}_{\text{type}}[\![\texttt{t}]\!]$ instead of `t`. To address this, we observe that the definition of $\mathcal{C}_{\text{env}}[\![-]\!]$ gives rise to the isomorphism:

$$\texttt{Index ts i t} \leftrightarrow \texttt{Index } \mathcal{C}_{\text{env}}[\![\texttt{ts}]\!] \texttt{ i } \mathcal{C}_{\text{type}}[\![\texttt{t}]\!]$$

so that we could equivalently write:

m : `Index` $\mathcal{C}_{\text{env}}[\![\texttt{ts}]\!]$ `i` $\mathcal{C}_{\text{type}}[\![\texttt{t}]\!]$ $\Rightarrow$ `SomeIndex ts'` $\mathcal{C}_{\text{type}}[\![\texttt{t}]\!]$

which is, in fact:

```
MapT C_env[[ts]] (SomeIndex ts')
```

thus the complete type of `cc` is:

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

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:

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

In practice, it is necessary for `fvs` to actually examine the type context `ts`, and we have in fact:

```
fv :: CtxRep ts -> ExpKb ts t -> MapT ts BoolT
```

where `CtxRep ts` reifies the type context `ts` as a Haskell value.

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]])
```

While this type captures the essence of what $mkEnv$ does, the index-mangling it performs rises slight complications. For one, the local variables map (m) and the environment ($j_0, \ldots, j_{n-1}$) grow in opposite directions as the recursion proceeds (c.f. the case $mkEnv\ (True\ :\ bs)\ j$). This is not harmful, but it takes a little extra machinery to track the way indices are appended to the environment. In terms of de Bruijn contexts, this means adding a binding "outside" a term, thus leaving intact an existing context where $i_0, \ldots, i_{n-1}$ are in scope while bringing into scope and extra index i_n. We reify such context extensions with the following type:

```
data Append ts s ts_s where
  A0 :: Append () s (s, ())
  As :: Append ts s ts_s -> Append (t, ts) s (t, ts_s)
```

which is introduced along with the fresh index (i_n):

```
newIndex :: CtxRep ts -> TypeRep s
            -> ∃ts_s, i. (Append ts s ts_s,
                          CtxRep ts_s,
                          Index ts_s i s)
```

and using which we can append to the environment:

```
appendM :: Append ts t ts_t -> MapT ts c -> c t
       -> MapT ts_t c
```

Also involved is the weakening of the already generated indices to fit into the extended context:

```
weaken :: Append ts s ts_s -> Index ts i t
       -> Index ts_s i t
```

Another implementation detail relates to the parameter (j) to $mkEnv$ which keeps track of the current index in the free variable list. An elegant way to do it is to construct beforehand a list of indices $[i_0, \ldots, i_{n-1}]$, and have $mkEnv$ recurse on this structure

[3] In practice, we use `data SomeIndex ts t = SomeIndex (Index ts i t)`

Hoisted programs, with variable names (L_H):

(*programs*)
p ::= letrec $\ell_0 = e_0$
$\vdots$
$\ell_{n-1} = e_{n-1}$
in e

(*exps*)
e ::= ... | ℓ_n

Hoisted programs, with de Bruijn indices (L_H^b):

(*programs*)
p ::= letrec e_0
$\vdots$
e_{n-1}
in e

(*indices*)
j ::= ... | ℓ_n

Figure 9. Target language of the hoisting transformation with variable names (left) and de Bruijn indices (right)

$$
\begin{array}{lcl}
collect\ \ell_m\ j & = & ([], j) \\
\\
collect\ \ell_m\ (\lambda\ e) & = & ([\lambda\ e', e_{m+1}, \ldots, e_n], \ell_m) \\
\quad \text{where } ([e_{m+1}, \ldots, e_n], e') = collect\ \ell_{m+1}\ e & & \\
\\
collect\ \ell_m\ (\mathsf{let}\ e_1\ \mathsf{in}\ e_2) & = & ([e_m, \ldots, e_n, e_{n+1}, \ldots, e_p], \\
 & & \quad \mathsf{let}\ e_1'\ \mathsf{in}\ e_2') \\
\quad \text{where } ([e_m, \ldots, e_n], e_1') = collect\ \ell_m\ e_1 & & \\
\qquad ([e_{n+1}, \ldots, e_p], e_2') = collect\ \ell_{n+1}\ e_2 & & \\
\ldots & &
\end{array}
$$

$$
\begin{array}{l}
hoist\ e = \mathsf{letrec}\ e_0 \\
\qquad\qquad \vdots \\
\qquad\qquad e_{n-1} \\
\qquad \mathsf{in}\ e' \\
\text{where } ([e_0, \ldots, e_{n-1}], e') = collect\ \ell_0\ e
\end{array}
$$

Figure 10. Hoisting transformation (transforms L_C^b into L_H^b)

simultaneously with the free variables bitmap $[b_0, \ldots, b_{n-1}]$. The list of indices has type:

```
MapT ts (SomeIndex ts)
```

and is easily constructed. Finally, the complete type of `mkEnv` is:

```
mkEnv :: CtxRep ts0           -- complete context
      -> CtxRep ts            -- partial context
      -> MapT ts BoolT        -- free variables
      -> MapT ts (IndexT ts0) -- indices i0,i1...
      -> CtxRep env0          -- original environment
      -> ∃Env. (CtxRep env,   -- extended environment
                MapT ts  (IndexT env),  -- m
                MapT env (IndexT ts0))  -- j0,j1...
```

where `ts0` is the de Bruijn context of the source term, `ts` is that part of the context with remains to be processed, `env0` is the part of the environment that has already been constructed, and `Env` is the type of the environment that is produced.

4. Hoisting

Hoisting and recursion The choice of target language may have a sensible impact on the way hoisting is performed. In Section 2.3, the fact that L_C has no recursive let construct forces us to nest the top-level let-bindings in a specific order, consistent with the "sub-term" relation among λ-abstractions in the source program. For instance, in Fig. 5, we could not have introduced ℓ_1 prior to ℓ_0, because ℓ_1 actually refers to ℓ_0 (thus in effect turning the source program upside down.) In a realistic compiler producing code with mutually recursive let definitions, this would not be an issue.

When compiling a source language with mutually recursive functions, hoisting amounts to merging all sets of mutually recursive functions into a single set. Here, we take the middle ground and show the compilation of our non-recursive language L_S^b into a recursive variant of L_C^b.

The target language is shown in Fig. 9. It extends L_C (or L_C^b) with a syntactic category of *programs*, providing a top-level letrec construct. The language is shown in both named variables (L_H) and de Bruijn form (L_H^b); we develop the hoisting transformation over the latter, the former being shown for illustration purpose only. The letrec construct introduces a number of variables $\ell_0, \ldots, \ell_{n-1}$; the scope of all those variables spans the body of all the letrec-bindings ($e_0, \ldots, e_{n-1}$) plus the program body (e). In the de Bruijn formulation, $\ell_0, \ldots, \ell_{n-1}$ form a new 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.

4.1 Implementation

We first describe a program representation for L_H^b, 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 L_H^b 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'll have to track thre type of the bundle of functions as it grows when recursive calls to $collect$ are made.

Importantly, a term is added to the *end* of the bundle. We already know how to represent this with types: we'll use the type

Append from Section 3.3. Here traversing a term may introduce multiple bindings (that is, when a term has multiple λ-abstractions as subterms.) To track the effect of appending an arbitrary number of terms to the bundle, we define a type that aggregates a number of Append's:

```
data Ext ts0 ts ts' where
  E0 :: Ext ts () ts
  Es :: Append ts0 s ts1 -> Ext ts1 ts ts'
          -> Ext ts0 (s, ts) ts'
```

A term of type `Ext ts (t0, (t1, ... (tn, ()) ...)) ts'` is a witness of the fact that appending types `t0, t1 ... tn` to the type `ts` yields type `ts'`. The implementation of *collect* is typed as follows:

```
collect ::
     CtxRep ts0
  -> CtxRep bs
  -> ExpB bs t        -- source term
  -> (∃ts, ts'.
        Ext ts0 ts ts',
        Ext bs ts' bsts',
        Tuple ts' ts,     -- the rest of the tuple
        ExpB bsts' t)     -- converted term
```

where the type variables –

- `ts0` reflects the type of the λ-abstractions already assigned to indices ($e_0 \ldots e_{m-1}$)
- `bs` is the de Bruijn context of the expression being converted (e),
- `ts` reflects the types of the λ-abstractions that are sub-terms of e and have been added to the tuple ($e_m \ldots e_n$),
- `ts'` reflects the types of $e_0 \ldots e_n$, and
- `bsts'` is the de Bruijn context of the converted expression: it adds binders $\ell_0 \ldots \ell_n$ to the e's original context.

Notably, the tuple of functions that is returned by *collect* is "partial": it consists of expressions e_m through e_n; however, expressions in this tuple are put in the complete context, with binders ℓ_0 through ℓ_n in scope. In the case of a source expression with multiple immediate sub-terms, such as let, *collect* must combine segments of the bundle together into a larger segment, and perform weakening on expressions typed in lesser contexts to end up with a well typed bundle of terms.

Finally, *hoist* simply takes a closed L_C^b term and produces a L_H^b program:

```
hoist :: ExpC () t -> Program t
```

5. Converting HOAS to de Bruijn indices

Our compiler front-end (which performs type-checking and CPS conversion) uses HOAS as its primary program representation. Having found de Bruijn better suited for closure conversion, we were faced with the task of converting HOAS to de Bruijn form. We report here the technique we applied for doing so.

As we expect this section to be of interest mainly to readers already familiar with HOAS programming, we assume familiarity with the techniques involved; the unfamiliar reader is referrer to [23] for a comprehensive background.

We illustrate the conversion to de Bruijn with the case of let-expressions, resuming the example from Section 3.1. 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); indeed the heart of the solution lies in picking β judiciously. Since the intent here is to re-produce the term in de Bruijn form, we'll have something of the form:

$$\beta\ \texttt{t} = \ldots \texttt{ExpB ts t} \ldots$$

where `ExpB ts t` is the type for de Bruijn terms in explicit type context `ts`.

In essence, the conversion to de Bruijn form introduces indices in place of variable occurrences. By nature, a de Bruijn index reflects the number of binders introduced between a variable occurrence and its corresponding binding occurrence. In terms of our concrete representation, the index measures the "difference" between the static context `ts` at the binding occurrence (that is, "outside" the let) and the context `ts'` where the variable occurs. Thus, a solution is to parameterize the result by the static context (reified as a value):

$$\beta\ \texttt{t} = \forall \texttt{ts. CtxRep ts -> ExpB ts t}$$

Now, the part that "does the work" inspects the two contexts `ts` and `ts'` and forms an index accordingly:

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

For `mkIndex` to succeed, `ts'` must actually be an extension of the type context `(t, ts)`, in the sense that new binders may have been introduced between the initial context and that in which appears the variable. Although it is indeed expected to always be the case, the types we use do not statically guaranty it; to remedy this, we need to compare the part of `ts'` that must match `(t, ts)`, reified as Haskell terms.

5.1 Fine points

The fact that index formation involves explicitly *comparing* segments of type contexts is not completely satisfactory. This, after all, amounts to testing rather than 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.

6. Related work

Closure conversion is a well-studied problem, both from a performance point of view [17], as well as its interaction with types [9, 10]. 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 [22] and FLINT [18, 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 de Bruijn indices and relies on type equality proofs in Haskell.

In [1], 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 de Bruijn indices, and their implementation language is more experimental. In a similar vein, Linger and Sheard [8] 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 [4], we showed the CPS phase of our compiler, where the distinguishing feature is the use of a term representation based on HOAS.

In [7], 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 [3], Fegaras and Sheard show how to handle higher-order abstract syntax, and in [23], 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 [24, 2, 20]. Their interaction with type classes is a known problem in GHC and a possible solution was proposed in [21].

7. Experience and Future work

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

Early on, we tried to use type classes to encode type-level functions as well as various proof objects. This would have helped us by letting the type checker infer more of the type annotations and hence leave us with a cleaner code more focused on the actual algorithm than on the type preservation proof. Sadly we bumped into serious difficulties due to the fact that the current version of GHC is not yet able to properly handle tight interactions been GADTs and functional dependencies. This limitation should hopefully be lifted shortly in a GHC near you, at which point we will definitely want to revisit this option.

GADTs We successfully and extensively use GADTs, but some of those uses are not quite satisfactory:

- GADTs are manipulated at runtime and thus incur a potentially significant performance cost. Laziness may help, as may GHC's optimizer, but we expect that GADTs which only encode proofs will be less efficient than if they were encoded with, say, type classes.
- Since Haskell is happy to allow constructing non-terminating objects, its corresponding logic is unsound. This means that representing proof terms as GADTs is not very satisfactory since the proof term may be $\bot$. Maybe type classes would help here as well.
- Encoding type-level functions as relations represented as GADTs is cumbersome. Using type-classes would be better, especially since the functional dependency could be checked by the type checker rather than having to write a proof term for it. But even better would be for Haskell's type system to provide type-level functions natively.

Future work Of course we intend to add more compilation phases, such as optimization and register allocation, to make it a more realistic compiler. 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 and to reduce the amount of code dedicated to proofs.

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 [5]: 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.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Princi-*

ples of Programming Languages, pages 42–54, New York, NY, USA, January 2006. ACM Press.

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

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

[10] 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.

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

[18] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.