Singleton types here
Singleton types there
Singleton types everywhere

Stefan Monnier David Haguenauer
Université de Montréal
{monnier,haguenad}@iro.umontreal.ca

Abstract

Singleton types are often considered a poor man's substitute for dependent types. But their generalization in the form of GADTs has found quite a following. The main advantage of singleton types and GADTs is to preserve the so-called *phase distinction*, which seems to be so important to make use of the usual compilation techniques.

Of course, they considerably restrict the programmers, which often leads them to duplicate code at both the term and type levels, so as to reflect at the type level what happens at the term level, in order to be able to reason about it.

In this article, we show how to automate such a duplication while eliminating the problematic dependencies. More specifically, we show how to compile the Calculus of Constructions into λ_H, a non-dependently-typed language, while still preserving all the typing information. Since λ_H has been shown to be amenable to type preserving CPS and closure conversion, it shows a way to preserve types when doing code extraction and more generally when using all the common compiler techniques.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Dependent types; D.3.4 [*Programming Languages*]: Processors—Compilation; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure

General Terms Algorithms, Languages, Verification

Keywords Dependent types, singleton types, certified compilation

1. Introduction

Compilation of dependently-typed languages has proved difficult. For example, it took a long time until someone finally figured how to do CPS conversion of dependently-typed languages [Barthe et al., 1999]. To the best of our knowledge, it is still not known how to perform closure conversion for such a language. Of course, that did not stop people from compiling such languages. The general approach has been to perform *code extraction* [Letouzey, 2008], which tries to eliminate all the parts of the code that do not affect the actual end result but instead only participate in the proofs and types. In other words, code extraction ends up throwing away some or all of the type information.

We show in this article a transformation which compiles the Calculus of Constructions [Coquand and Huet, 1988] into a non-dependently-typed language, namely λ_H [Shao et al., 2002]. After this transformation, the code is amenable to all the usual compilation techniques, and indeed performing the equivalent of code extraction can then be done by applying known optimization techniques such as elimination of unused arguments or hoisting code out of conditionals. The novel part here is that all this can be done without losing any type information, such that the end code can still be type-checked.

λ_H was designed as an internal language for compilers that want to generate certified binaries, and has been shown to be able to handle these needs very well: the original article [Shao et al., 2002] shows how to perform the usual CPS and closure conversion while preserving types; then League and Monnier [2006] shows essentially that λ_H can be seen as a type system for $\lambda ink_{\subsetneq}$ [Fisher et al., 2000], and is hence a good target language for compilation of most object-oriented features; and Monnier [2004] showed how a variant of λ_H extended with regions can be used to write a type-preserving generational garbage collector. Still, λ_H only provides singleton types as a poor man's substitute to dependent types which, while apparently sufficient for those particular cases, might lead one to believe that it is not up to the task of handling truly dependent types. We here show this belief to be unfounded.

Of course it is not a complete surprise: one of the reasons is, of course, that λ_H includes the Calculus of Constructions as a subset of its type language, so while dependent typing is not supported for the actual computational terms, it is fully available at the level of types. Another reason is that anybody who has used GADTs or singleton types has probably learned that whenever dependent types seem necessary, you can circumvent the problem by duplicating code. For example, if you need your type system to understand what your *sort* function does to its input, "all" you need to do is to make sure the inputs are singleton-typed, duplicate your *sort* computational function into a type-level *Ssort* function, and then show that *sort* has type $\forall l : List\ \alpha . Slist\ l \rightarrow Slist\ (Ssort\ l)$. Such code duplication is obviously unpleasant and frustrating, all the more so since it is very mechanical. Automatically reflecting the computation function into a type function is not feasible in general since the computation language includes features such as side effects which are not available at the level of types, but automatically reifying a type level function into a computational function of a type like the one above is, on the other hand, quite doable, and is precisely what our transformation does. Our contributions are:

- An algorithm that compiles the Calculus of Constructions to a non-dependently-typed language, while preserving all the type information. By that we mean that the function mapping the type of the input code to the type of the output code is injective.
- Coupled with the CPS conversion and closure conversion algorithms already existing for the target language, this is to our best knowledge the first solution to the problem of performing a type preserving closure conversion for a dependently-typed language.
- Show that the "poor man's substitute for dependent types", although obviously not as elegant, can be just as powerful as truly dependent types, in the sense that its types can express any constraint expressible with dependent types. Maybe they would be more appropriately named "hard-working man's substitute for dependent types".
- Demonstrate that a language with a phase distinction, like λ_H, could automatically reify type definitions into the computation language, saving the poor hard-working man from having to duplicate his code by hand.

The rest of this article is structured as follows: Sec. 2 presents the languages and techniques on which we build our algorithm. Sec. 3 shows the algorithm proper. Sec. 4 presents the formal properties and proof sketches. Sec. 5 generalizes the algorithm to apply to the compilation of a Pure Type System. Sec. 6 then concludes with a discussion of related and future work.

2. Background

Here, we quickly review the languages and techniques used in this article.

2.1 Pure Type Systems

A Pure Type System (PTS) [Barendregt, 1991] is a convenient way to specify a typed λ-calculus. A term in a PTS has the following syntax:

$$(\textit{ptm})\ A, B ::= s \mid X \mid \lambda X{:}A.B \mid A\ B \mid \Pi X{:}A.P$$

$\boxed{\Delta \vdash A : B}$: term A has type B in context Δ

$$\text{PTS-AXIOM}\ \frac{A{:}B \in \mathcal{A}}{\bullet \vdash A : B} \qquad \text{PTS-VAR}\ \Delta \vdash X : \Delta(X) \qquad \text{PTS-WEAK}\ \frac{\Delta \vdash A' : s \qquad \Delta \vdash A : B}{\Delta, X{:}A' \vdash A : B}$$

$$\text{PTS-FUN}\ \frac{\Delta, X{:}A \vdash B_1 : B_2 \qquad \Delta \vdash \Pi X{:}A.B_2 : s}{\Delta \vdash \lambda X{:}A.B_1 : \Pi X{:}A.B_2}$$

$$\text{PTS-APP}\ \frac{\Delta \vdash A_1 : \Pi X{:}A_2.B_2 \qquad \Delta \vdash B_1 : A_2}{\Delta \vdash A_1\ B_1 : B_2[B_1/X]}$$

$$\text{PTS-PROD}\ \frac{\Delta \vdash A : s_1 \qquad \Delta, X{:}A \vdash B : s_2 \qquad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X{:}A.B : s_2}$$

$$\text{PTS-CONV}\ \frac{\Delta \vdash A : B_1 \qquad B_1 \approx_\beta B_2 \qquad \Delta \vdash B_2 : s}{\Delta \vdash A : B_2}$$

Figure 1. Typing rules for PTS

$$\begin{array}{lll} (\textit{kscm}) & u & ::= \Pi t{:}\kappa.u \mid \Pi k{:}u.u \mid \mathsf{Kind} \\ (\textit{kind}) & \kappa & ::= k \mid \Pi t{:}\kappa.\kappa \mid \Pi k{:}u.\kappa \\ & & \quad \mid \lambda t{:}\kappa.\kappa \mid \lambda k{:}u.\kappa \mid \kappa\ \tau \mid \kappa\ \kappa \\ (\textit{type}) & \tau & ::= t \mid \lambda t{:}\kappa.\tau \mid \lambda k{:}u.\tau \\ & & \quad \mid \tau\ \tau \mid \tau\ \kappa \end{array}$$

Figure 2. Syntax of our source language, CC

where X ranges over variables, and s are predefined sorts. The associated typing rules are given in Fig. 1. To define a particular PTS, you only have to specify the set $\mathcal{S}$ of sorts, their typing axioms $\mathcal{A}$, and the allowed forms of abstraction and quantification $\mathcal{R}$. For example System F [Girard, 1972, Reynolds, 1974] can be specified as follows:

$$\begin{array}{rcl} \mathcal{S} & = & \mathsf{Type}, \mathsf{Kind} \\ \mathcal{A} & = & \mathsf{Type} : \mathsf{Kind} \\ \mathcal{R} & = & (\mathsf{Type}, \mathsf{Type}), (\mathsf{Kind}, \mathsf{Type}) \end{array}$$

Which says that there are 2 sorts, one for types and one for kinds (in traditional presentations of System F kinds are usually elided since there is only one object of type Kind, namely Type, but here we cannot take this shortcut). The most interesting part is $\mathcal{R}$ which says that λ-abstractions can take terms to terms (the traditional λ) or types to terms (the usual Λ).

2.2 The Calculus of Constructions

The source language we will compile is the Calculus of Constructions (CC) [Coquand and Huet, 1988]. This is a higher-order, dependently-typed λ-calculus which is used here as a core representative of dependently-typed languages. Its syntax is shown in Fig. 2. As we can see, the language has three layers: terms, types, and kinds. But because of how we use this language in this article, we prefer to shift those terms by one level, so as to call the three layers: types, kinds, and kind schemas.

Rather than give the typing rules for CC, which are very repetitive, because of the four λ and corresponding four Π and four applications, we will simply say that CC can also be defined as a Pure Type System:

$$\begin{array}{rcl} \mathcal{S} & = & \mathsf{Kind}, \mathsf{Kscm} \\ \mathcal{A} & = & \mathsf{Kind} : \mathsf{Kscm} \\ \mathcal{R} & = & (\mathsf{Kind}, \mathsf{Kind}), (\mathsf{Kscm}, \mathsf{Kind}), \\ & & (\mathsf{Kind}, \mathsf{Kscm}), (\mathsf{Kscm}, \mathsf{Kscm}) \end{array}$$

where each element of $\mathcal{R}$ corresponds to a λ in the stratified syntax used in Fig. 2.

This is a very powerful calculus in which we can encode constructive proofs of very complex propositions. Many proof assistants are actually based on a calculus reminiscent of CC, although usually extended in various ways. So this is a good baseline to show that our technique can handle realistic languages. Most notably missing here are inductive definitions and dependent elimination. Some inductive definitions can be expressed, using the so-called *impredicative encoding*, since CC, contrary to many of its siblings, is not predicative. Yet, the lack of inductive definitions and accompanying dependent elimination in our source language is a significant limitation we intend to lift.

2.3 TL and λ_H

Shao et al. [2002] presented a computation language called λ_H whose purpose was to serve as a typed intermediate language in certifying compilers. The languages that such compilers were expected to handle were left open, although most examples at that time concentrated on compiling traditional functional programming languages. Despite the apparent simplicity of the kind of type

$$
\begin{array}{lll}
(\textit{kscm}) & u & ::= \Pi t{:}\kappa.u \mid \Pi k{:}u.u \mid \mathsf{Kind} \mid z \\
(\textit{kind}) & \kappa & ::= k \mid \Pi t{:}\kappa.\kappa \mid \Pi k{:}u.\kappa \mid \Pi z{:}\mathsf{Kscm}.\kappa \\
 & & \quad \mid \lambda t{:}\kappa.\kappa \mid \lambda k{:}u.\kappa \mid \kappa\ \tau \mid \kappa\ \kappa \\
 & & \quad \mid \mathsf{Ind}(k{:}\mathsf{Kind})\{\vec{\kappa}\} \mid \mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa}\} \\
(\textit{type}) & \tau & ::= t \mid \lambda t{:}\kappa.\tau \mid \lambda k{:}u.\tau \mid \lambda z{:}\mathsf{Kscm}.\tau \\
 & & \quad \mid \tau\ \tau \mid \tau\ \kappa \mid \tau\ u \\
 & & \quad \mid \mathsf{Ctor}(i, \kappa) \mid \mathsf{Elim}[\kappa, \kappa](\tau)\{\vec{\tau}\}
\end{array}
$$

Figure 3. Stratified syntax of TL

$$
\begin{array}{lll}
(\textit{exp}) & e & ::= x \mid e\ e \mid e[A] \\
(\textit{fun}) & f & ::= \mathsf{fn}\ x{:}\tau.e \mid \Lambda X{:}A.f
\end{array}
$$

Figure 4. Syntax of λ_H

system needed to handle such languages, λ_H provided a very powerful type language (called TL), which is a superset of CC, where the main extension is the addition of inductive definitions. This power was mostly justified at the time by the needs of fully reflexive intensional type analysis [Trifonov et al., 2000], as well as by the needs of the very late phases of the compiler, where optimizations such as array bounds elimination can require non-trivial expressive power to explain in the type annotations why the resulting code is still safe. Subsequent work has shown that λ_H is actually also a good choice when compiling other kinds of languages, thanks to the flexibility of its type language.

The design of λ_H was driven on one side by the desire to offer a powerful type system where complex properties and proofs could be expressed and manipulated, and on the other side by the constraint to preserve the *phase distinction* between computations and types, meaning that types and computations live in clearly distinct worlds that live in different phases: types at compile-time and computations at run-time. This constraint means both that computations cannot depend on types and that types cannot depend on computations; the first is important in order to be able to use classical compilation techniques, and the second is important for decidable type checking and to be able to preserve type information even as the computational part of the program goes through extensive transformations that may even require changing the computational language.

The language in which types are written in λ_H is called TL and its syntax is shown in Fig. 3, in stratified form. We will also use PTS notations to refer to it, when convenient. As can be seen, it is a superset of CC, where a fifth abstraction, from kind schemas to types, has been added, together with inductive definitions, whose constructs are:

- $\mathsf{Ind}(k{:}\mathsf{Kind})\{\vec{\kappa}\}$: the inductive definition as such. It is in many ways similar to a datatype or a GADT definition.
- $\mathsf{Ctor}(i, \kappa)$: the constructor for an object of inductive type. κ is the inductive definition and i is the index of the constructor.
- $\mathsf{Elim}[\kappa, \kappa](\tau)\{\vec{\tau}\}$: the elimination construct for inductive definitions, it can be thought of as an induction scheme, or a case analysis construct.

While inductive definitions in TL are a very welcome addition that makes manipulating proofs a lot more convenient, they are also used directly in the definition of λ_H itself. More specifically, the set of types that classify valid λ_H programs is defined as an inductive definition in TL, usually written in the following way, using a Coq-inspired syntax:

$$
\begin{array}{lll}
\mathsf{Inductive}\ \textit{Nat} : \mathsf{Kind} := O & : & \textit{Nat} \\
\quad \mid S & : & \textit{Nat} \to \textit{Nat} \\
\\
\mathsf{Inductive}\ \Omega : \mathsf{Kind} := \textit{snat} & : & \textit{Nat} \to \Omega \\
\quad \mid \twoheadrightarrow & : & \Omega \to \Omega \to \Omega \\
\quad \mid \forall_{\mathsf{Kind}} & : & \Pi k{:}\mathsf{Kind}.(k \to \Omega) \to \Omega \\
\quad \mid \forall_{\mathsf{Kscm}} & : & \Pi z{:}\mathsf{Kscm}.(z \to \Omega) \to \Omega
\end{array}
$$

So the type of a λ_H function has the form $\twoheadrightarrow\ \tau_1\ \tau_2$, although we will usually write it $\tau_1 \twoheadrightarrow \tau_2$ instead for convenience. Similarly a polymorphic λ_H function will have a type of the form $\forall_s\ A\ (\lambda X{:}A.\tau)$, but we will usually write it as $\forall_s X{:}A.\tau$.

$\boxed{\Delta; \Gamma \vdash e : \tau}$: in type environment Δ and value environment Γ, e has type τ.

$$
\text{E-VAR}\ \ \Delta; \Gamma \vdash x : \Gamma(x)
\qquad
\text{E-FUN}\ \ \frac{\Delta \vdash \tau_1 : \Omega \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \mathsf{fn}\ x{:}\tau_1.e : \tau_1 \twoheadrightarrow \tau_2}
$$

$$
\text{E-APP}\ \ \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \twoheadrightarrow \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}
$$

$$
\text{E-TFUN}\ \ \frac{\Delta \vdash A : s \qquad \Delta, X{:}A; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda X{:}A.e : \forall_s X{:}\kappa.\tau}
$$

$$
\text{E-TAPP}\ \ \frac{\Delta; \Gamma \vdash e : \forall_s X{:}B.\tau \qquad \Delta \vdash A : B}{\Delta; \Gamma \vdash e[A] : \tau[A/X]}
$$

$$
\text{E-CONV}\ \ \frac{\Delta; \Gamma \vdash e : \tau_1 \qquad \tau_1 \approx_\beta \tau_2 \qquad \Delta \vdash \tau_2 : \Omega}{\Delta; \Gamma \vdash e : \tau_2}
$$

Figure 5. Typing rules for λ_H

The syntax of λ_H is shown in Fig. 4. λ_H as presented in [Shao et al., 2002] has many other constructs and types defined, but since we will not use them here, we took the liberty to remove them. As it happens, other than the above, we will not use any other inductive definitions either, and we will not even use any elimination construct on inductive definitions. The typing rules of λ_H are given in Fig. 5. Compared to the Pure Type system typing rules, the only difference is that the environment is now split into Δ which holds the kinds of all type variables and Γ which holds the types of all the term variables.

A key element of λ_H is that its type language TL needs to stand on its own and be close to preexisting systems, since we need it to satisfy non-trivial meta-properties such as consistency, so we would rather take advantage of the work of others in this respect. The connection between TL and the term language is made on the one hand by the inductive definition of Ω and on the other by the term typing rules.

2.4 Reflecting terms and reifying types

Singleton types are based on the idea of duplicating some term construct to the type level (*reflecting* them into types). For example, the canonical case of singleton types, the singleton integers, works by reflecting the integers provided at the term level into types, and then indexing the type of singleton integers with that reflected notion of integers. We replace:

$$n : \mathsf{int}$$

with

$$\breve{n} : \mathsf{sint}\ \hat{n}$$

where $\hat{n}$ is the reflection at the type level of the term-level integer n, $\breve{n}$ is the new singleton-typed integer constant, and sint is the new type of singleton integers, indexed with a type-level integer. We can do the same with other datatype definitions, and if you try to find a pattern, you will see that converting a non-singleton data

$$e : \mathsf{MyType}$$

into its singleton typed equivalent will basically always turn it into something morally equivalent to:

$$\breve{e} : \mathsf{S}\langle\mathsf{MyType}\rangle\ \hat{e}$$

where $\hat{e}$ is the reflection of e at the level of types, $\breve{e}$ is the new singleton-typed object, and $\mathsf{S}\langle\mathsf{MyType}\rangle$ is the new definition of MyType, where the only difference is the addition of the $\hat{e}$ indexing.

The way this duplication works is by creating mirror images of the objects we manipulate, which are then available both at the term level and at the type level, so that, depending on where we need to refer to them, we can choose to either use the term representation, or the type representation, thus breaking the need for types to refer to terms.

But once you have reflected your data into your types, you will soon find that you also want to reflect your functions. E.g. we do not want to assign the following type to our addition function, since it would prevent most useful forms of reasoning in the types about the arithmetic operations we perform on terms:

$$+ : \forall n, m{:}\mathsf{int.sint}\ n \to \mathsf{sint}\ m \to (\exists o{:}\mathsf{int.sint}\ o)$$

instead we want to assign it the type:

$$\breve{+} : \forall n, m{:}\mathsf{int.sint}\ n \to \mathsf{sint}\ m \to (n \hat{+} m)$$

where $\hat{+}$ is a reflection of the addition, i.e. the type-level addition. If you squint hard enough, you will see that the above type can be treated as being of the form $\mathsf{S}\langle\mathsf{int} \to \mathsf{int} \to \mathsf{int}\rangle\ \hat{+}$. Whether such an addition function deserves the name "singleton" is up for debate, since there may be different implementations of such a function, but they will all behave in the same way: they might be intensionally different, but extensionally identical. For the sake of this article, we will consider it as a form of singleton typing.

More generally, a function f whose original signature is $\tau_1 \to \tau_2$, and which the user wants to turn into a singleton typed function, will have to have a type of the form:

$$\breve{f} : \forall x{:}\tau_1.\mathsf{S}\langle\tau_1\rangle\ x \to \mathsf{S}\langle\tau_2\rangle\ (\hat{f}\ x)$$

Until now we have taken the point of view of the user who has a classically-typed piece of data or function and wants to reflect it to the type level. But if you look again at the examples above, the original classical *types* (such as int, or τ_1) get turned into *kinds* by the change to singleton types. So we start with 2 levels (terms and types), and end with 3, so it is up to the user to interpret this transformation as taking an e and returning either an equivalent $\breve{e}$ (plus a $\hat{e}$ reflection), or inversely returning an equivalent $\hat{e}$ (plus a $\breve{e}$ reification). As it turns out, the second interpretation is easier to follow, because indeed, the shape and type of $\hat{e}$ is really the same as that of e, whereas the shape and type of $\breve{e}$ is significantly different from the one of e.

3. Reifying all types to terms

As shown above, we fundamentally need 3 operations: one to turn e into $\hat{e}$, another to get $\breve{e}$ and the last one to find the type $\mathsf{S}\langle\tau\rangle$. As mentioned, a source term e is virtually identical to its destination type $\hat{e}$, which is why we have defined CC as made of types, kinds, and kind schemas rather than terms, types, and kinds. This way we can conflate e and $\hat{e}$, and we are left with only two transformations to define: one to reify e into $\breve{e}$ and the other to reify the kind κ of the type e (remember, we shifted levels) into a type "constructor" $\mathsf{S}\langle\kappa\rangle$.

Before showing the actual algorithm, we will try to give some intuition about how we can get there.

3.1 Intuition

We will denote our reification translation from CC to λ_H as $\mathcal{C}$. This function will basically take a term e and return the corresponding $\breve{e}$. The main entry point is to be applied to a CC type τ, and, since some other parts of $\mathcal{C}$ will apply to kinds and to kind schemas, we call the main entry point $\mathcal{C}_t$ and the other ones $\mathcal{C}_k$ and $\mathcal{C}_u$. As described above, when applied to a λ-term f of type $\Pi t{:}\kappa_1.\kappa_2$ it will return something of type:

$$\mathcal{C}_t\ [\![\lambda t{:}\kappa_1.f\ t]\!] : \forall_{\mathsf{Kind}} t{:}\kappa_1.(\mathcal{C}_k\ [\![\kappa_1]\!])t \twoheadrightarrow (\mathcal{C}_k\ [\![\kappa_2]\!])(f\ t)$$

where we have renamed S to $\mathcal{C}_k$ since it is the part of $\mathcal{C}$ that applies to kinds. Notice that there is a good reason to use the same name, since just like $\mathcal{C}_t$ reifies types into terms, $\mathcal{C}_k$ reifies kinds into types. Moreover there is a one to one correspondence between the input kind and the output type: $\mathcal{C}_k$ creates singleton type constructors which are themselves singleton-kinded!

But let us come back to $\mathcal{C}_t$: given the above type, we know that the reification of a λ term will look like:

$$\mathcal{C}_t\ [\![\lambda t{:}\kappa_1.f\ t]\!] = \Lambda t{:}\kappa_1.\mathsf{fn}\ x{:}(\mathcal{C}_k\ [\![\kappa_1]\!])t....$$

where the ... needs to have a type of the form $(\mathcal{C}_k\ [\![\kappa_2]\!])(f\ t)$, which is luckily the type of $\mathcal{C}_t\ [\![f\ t]\!]$, since the type of $f\ t$ is κ_2 and we generally want to have $\mathcal{C}_t\ [\![\tau]\!] : (\mathcal{C}_k\ [\![\kappa]\!])\ \tau$. So the rule looks like

$$\mathcal{C}_t\ [\![\lambda t{:}\kappa.\tau]\!] = \Lambda t{:}\kappa.\mathsf{fn}\ x{:}(\mathcal{C}_k\ [\![\kappa]\!])t.\mathcal{C}_t\ [\![\tau]\!]$$

Now, what we have above is that the original argument t is duplicated into t and x, where t is the (unchanged) type-level representation of the original type t and x is its reified term-level representation. And we will want to keep track of which term variable corresponds to which type variable so that when we need to refer to t at the term level we can use x instead. So $\mathcal{C}_t$ needs an additional argument which keeps track of this mapping. We will call it c_t since it is a sort of $\mathcal{C}_t$ but specialized to only apply to type variables (for people accustomed to HOAS [Pfenning and Elliott, 1988], it is more like a scoped extension of the function $\mathcal{C}_t$, using a hypothetical judgment). I.e. we need to adjust the above rule as follows:

$$\mathcal{C}_t\ c_t\ [\![\lambda t{:}\kappa.\tau]\!] = \Lambda t{:}\kappa.\mathsf{fn}\ x{:}(\mathcal{C}_k\ [\![\kappa]\!])t.\mathcal{C}_t\ \{c_t, t \mapsto x\}\ [\![\tau]\!]$$
$$\mathcal{C}_t\ c_t\ [\![t]\!] = c_t(t)$$

Also, if we want $\mathcal{C}_t\ [\![\tau]\!] : (\mathcal{C}_k\ [\![\kappa]\!])\ \tau$, the above rule implies that

$$\mathcal{C}_k\ [\![\Pi t{:}\kappa_1.\kappa_2]\!] = \lambda f{:}(\Pi t{:}\kappa_1.\kappa_2).\forall_{\mathsf{Kind}} t{:}\kappa_1. (\mathcal{C}_k\ [\![\kappa_1]\!])t \twoheadrightarrow (\mathcal{C}_k\ [\![\kappa_2]\!])(f\ t)$$

If we now go back to the rule for $\mathcal{C}_t\ [\![\tau_1\ \tau_2]\!]$, the above indicates that it should look like the following for the rules to be self-consistent:

$$\mathcal{C}_t\ c_t\ [\![\tau_1\ \tau_2]\!] = (\mathcal{C}_t\ c_t\ [\![\tau_1]\!])[\tau_2](\mathcal{C}_t\ c_t\ [\![\tau_2]\!])$$

The other rules follow the same principle, tho adjusted for each particular form of λ abstraction. Additionally to those rules we will need to provide base rules for each of the built-in types:

$$\mathcal{C}_k\ [\![\mathsf{nat}]\!] = \mathit{Snat}$$

and of course, we need to figure out how to translate a kind variable k. This will require an additional argument c_k, that serves the same purpose as c_t but for kinds rather than for types: whenever we pass a kind (such as nat) as an argument to a function, we need to also

$\boxed{\mathcal{C}_t\ c_k\ c_t\ [\![\tau : \kappa]\!] : (\mathcal{C}_k\ c_k\ [\![\kappa]\!])\ \tau}$

$$
\begin{array}{lcl}
\mathcal{C}_t\ c_k\ c_t\ [\![t]\!] & = & c_t(t) \\
\mathcal{C}_t\ c_k\ c_t\ [\![\lambda t{:}\kappa.\tau]\!] & = & \Lambda t{:}\kappa.\mathsf{fn}\ x{:}(\mathcal{C}_k\ c_k\ [\![\kappa]\!])t.\mathcal{C}_t\ c_k\ \{c_t, t \mapsto x\}\ [\![\tau]\!] \\
\mathcal{C}_t\ c_k\ c_t\ [\![\lambda k{:}u.\tau]\!] & = & \Lambda k{:}u.\Lambda t{:}(\mathcal{C}_u\ [\![u]\!])k.\mathcal{C}_t\ \{c_k, k \mapsto t\}\ c_t\ [\![\tau]\!] \\
\mathcal{C}_t\ c_k\ c_t\ [\![\tau_1\ \tau_2]\!] & = & (\mathcal{C}_t\ c_k\ c_t\ [\![\tau_1]\!])[\tau_2]\ (\mathcal{C}_t\ c_k\ c_t\ [\![\tau_2]\!]) \\
\mathcal{C}_t\ c_k\ c_t\ [\![\tau\ \kappa]\!] & = & (\mathcal{C}_t\ c_k\ c_t\ [\![\tau]\!])[\kappa][(\mathcal{C}_k\ c_k\ [\![\kappa]\!])]
\end{array}
$$

$\boxed{\mathcal{C}_u\ [\![u : \mathsf{Kscm}]\!] : \Pi\_{:}u.\mathsf{Kind}}$

$$
\begin{array}{lcl}
\mathcal{C}_u\ [\![\mathsf{Kind}]\!] & = & \lambda k{:}\mathsf{Kind}.\Pi\_{:}k.\Omega \\
\mathcal{C}_u\ [\![\Pi t{:}\kappa.u]\!] & = & \lambda f{:}(\Pi t{:}\kappa.u).\Pi t{:}\kappa.(\mathcal{C}_u\ [\![u]\!])\ (f\ t) \\
\mathcal{C}_u\ [\![\Pi k{:}u_1.u_2]\!] & = & \lambda f{:}(\Pi k{:}u_1.u_2).\Pi k{:}u_1.\Pi t{:}(\mathcal{C}_u\ [\![u_1]\!])k. \\
 & & (\mathcal{C}_u\ [\![u_2]\!])\ (f\ k)
\end{array}
$$

$\boxed{\mathcal{C}_k\ c_k\ [\![\kappa : u]\!] : (\mathcal{C}_u\ [\![u]\!])\ \kappa}$

$$
\begin{array}{lcl}
\mathcal{C}_k\ c_k\ [\![k]\!] & = & c_k(k) \\
\mathcal{C}_k\ c_k\ [\![\mathsf{nat}]\!] & = & \mathsf{Snat} \\
\mathcal{C}_k\ c_k\ [\![\Pi t{:}\kappa_1.\kappa_2]\!] & = & \lambda f{:}(\Pi t{:}\kappa_1.\kappa_2).\forall_{\mathsf{Kind}} t{:}\kappa_1. \\
 & & (\mathcal{C}_k\ c_k\ [\![\kappa_1]\!])t \twoheadrightarrow (\mathcal{C}_k\ c_k\ [\![\kappa_2]\!])(f\ t) \\
\mathcal{C}_k\ c_k\ [\![\Pi k{:}u.\kappa]\!] & = & \lambda f{:}(\Pi k{:}u.\kappa).\forall_{\mathsf{Kscm}} k{:}u.\forall_{\mathsf{Kind}} t{:}(\mathcal{C}_u\ [\![u]\!])k. \\
 & & (\mathcal{C}_k\ \{c_k, k \mapsto t\}\ [\![\kappa]\!])(f\ k) \\
\mathcal{C}_k\ c_k\ [\![\kappa\ \tau]\!] & = & (\mathcal{C}_k\ c_k\ [\![\kappa]\!])\ \tau \\
\mathcal{C}_k\ c_k\ [\![\kappa_1\ \kappa_2]\!] & = & (\mathcal{C}_k\ c_k\ [\![\kappa_1]\!])\ \kappa_2\ (\mathcal{C}_k\ c_k\ [\![\kappa_2]\!]) \\
\mathcal{C}_k\ c_k\ [\![\lambda t{:}\kappa_1.\kappa_2]\!] & = & \lambda t{:}\kappa_1.\mathcal{C}_k\ c_k\ [\![\kappa_2]\!] \\
\mathcal{C}_k\ c_k\ [\![\lambda k{:}u.\kappa]\!] & = & \lambda k{:}u.\lambda t{:}(\mathcal{C}_u\ [\![u]\!])k. \\
 & & \mathcal{C}_k\ \{c_k, k \mapsto t\}\ [\![\kappa]\!]
\end{array}
$$

Figure 6. The compilation algorithm from CC to λ_H

pass the corresponding translation (e.g. *Snat*) and we have to record somewhere the correspondence between the two (type and kind) variables. So the structure from $\mathcal{C}_t$ gets reproduced in $\mathcal{C}_k$, which leads to the need to introduce $\mathcal{C}_u$ to map the type of κ to the type of $\mathcal{C}_k\ [\![\kappa]\!]$.

3.2 The translation algorithm

The complete algorithm is given is Fig. 6. The three functions $\mathcal{C}_t$, $\mathcal{C}_k$, and $\mathcal{C}_u$ show a lot of similarity, which in retrospect is fairly natural, but definitely was not expected originally. Fundamentally what this code tells us is that we should consider the type $\mathcal{C}_k\ [\![\kappa]\!]$ not only to be a singleton type constructor but also the single type of the singleton kind $(\mathcal{C}_u\ [\![u]\!])\kappa$.

Of course, while there are many similarities, there are also important differences:

- The argument c_t is not passed to $\mathcal{C}_k$ and $\mathcal{C}_u$: this is simply because these two functions operate at the level of types, so we will never want to find the singleton term variable corresponding to a particular type variable and will prefer to use the type variable instead. After all, that is the whole point of the exercise: not referring to terms from types.
- Similarly, c_k is not passed up to $\mathcal{C}_u$: contrary to the previous point, $\mathcal{C}_u$ is free to refer to types, so it could potentially make use of c_k, but it turns out that it never needs to. The reason is that it would only use it if it needed somewhere to duplicate a binding of the form $t : \kappa$, but that would again be a case where we are trying to duplicate a type into a term, while being in the realm of types where we prefer to refer to types than to terms.
- There is no c_u argument: given that we do not have kind schema variables, there would be no mapping to put inside.
- The type of $\mathcal{C}_u$ looks different from the rest: actually, this is just an illusion. It is identical if we replace $\Pi\_ : u.\mathsf{Kind}$ with $(\mathcal{C}_e\ [\![\mathsf{Kscm}]\!])\ u$ and then define $\mathcal{C}_e\ [\![\mathsf{Kscm}]\!] = \lambda z : \mathsf{Kscm}.\Pi\_{:}z.\mathsf{Kind}$.
- $\mathcal{C}_k$ does not duplicate the argument t when translating $\lambda t{:}\kappa_1.\kappa_2$: indeed, it does not, because that duplicate of t would have to be a term, which cannot exist in TL. Of course, not only it cannot duplicate t, but it also does not need to duplicate it, because, at the level of types, we will never want to refer to the term form of t and will prefer to use just t instead. This affects not only the $\mathcal{C}_k$ translation of $\lambda t{:}\kappa_1.\kappa_2$ but of course as well its translation of the corresponding function application, and the $\mathcal{C}_u$ translation of $\Pi t{:}\kappa.u$.

Discussion We said above that $\mathcal{C}_k\ [\![\kappa]\!]$ is the single type of the singleton kind $(\mathcal{C}_u\ [\![u]\!])\kappa$, but in fact this needs to be qualified: in reality, those kinds often are inhabited by more than one type, so while it may help to think of them as singleton kinds, they are not actually singleton. For example the kind of the type $\mathcal{C}_k\ [\![\mathsf{nat}]\!]$ is $\mathsf{nat} \to \Omega$, which is also the kind of $\mathsf{Snat} \circ S$ and many other types.

This points to the fact that there is some amount of flexibility in the translation of the base types like nat, and that it makes it possible to choose to lose some type information by simply changing the translation of the base types (and corresponding values, of course).

4. Formal properties

The main property of the algorithm is that it preserves types.

THEOREM 4.1 (Type preservation).
For all Δ, c_t, c_k such that
$\forall (t{:}\kappa) \in \Delta.\ c_t(t) = x \wedge x \notin \mathsf{Range}(c_t)$ and
$\forall (k{:}u) \in \Delta.\ c_k(k) = t \wedge t \notin \mathsf{Range}(c_k) \wedge t \notin \mathsf{Dom}(\Delta)$ then:

- for all τ and κ such that $\Delta \vdash \tau : \kappa$, then
  $(\mathcal{C}_\Delta\ c_k\ [\![\Delta]\!]); (\mathcal{C}_\Gamma\ c_k\ c_t\ [\![\Delta]\!]) \vdash \mathcal{C}_t\ c_k\ c_t\ [\![\tau]\!] : (\mathcal{C}_k\ c_k\ [\![\kappa]\!])\ \tau$
- for all κ and u such that $\Delta \vdash \kappa : u$, then
  $(\mathcal{C}_\Delta\ c_k\ [\![\Delta]\!]) \vdash \mathcal{C}_k\ c_k\ [\![\kappa]\!] : (\mathcal{C}_u\ [\![u]\!])\ \kappa$
- for all u such that $\Delta \vdash u : \mathsf{Kscm}$, then
  $(\mathcal{C}_\Delta\ c_k\ [\![\Delta]\!]) \vdash \mathcal{C}_u\ [\![u]\!] : \Pi\_{:}u.\mathsf{Kscm}$

This theorem relies on auxiliary functions $\mathcal{C}_\Gamma$ and $\mathcal{C}_\Delta$ defined as follow:

$$
\begin{array}{lcl}
\mathcal{C}_\Gamma\ c_k\ c_t\ [\![\bullet]\!] & = & \bullet \\
\mathcal{C}_\Gamma\ c_k\ c_t\ [\![\Delta, k{:}u]\!] & = & \mathcal{C}_\Gamma\ c_k\ c_t\ [\![\Delta]\!] \\
\mathcal{C}_\Gamma\ c_k\ c_t\ [\![\Delta, t{:}\kappa]\!] & = & (\mathcal{C}_\Gamma\ c_k\ c_t\ [\![\Delta]\!]), c_t(t) : (\mathcal{C}_k\ c_k\ [\![\kappa]\!])\ t \\
 & & \\
\mathcal{C}_\Delta\ c_k\ [\![\bullet]\!] & = & \bullet \\
\mathcal{C}_\Delta\ c_k\ [\![\Delta, t{:}\kappa]\!] & = & (\mathcal{C}_\Delta\ c_k\ [\![\Delta]\!]), t{:}\kappa \\
\mathcal{C}_\Delta\ c_k\ [\![\Delta, k{:}u]\!] & = & (\mathcal{C}_\Delta\ c_k\ [\![\Delta]\!]), k{:}u, c_k(k) : (\mathcal{C}_u\ [\![u]\!])\ k
\end{array}
$$

The proof is by induction over the typing derivations. It is as tedious as any, but does not encounter any major obstacle. It does require several lemmas shown below, which each follow the same pattern of induction over the typing or evaluation derivation. Those lemmas state that substitutions can be pushed through the $\mathcal{C}_k$, and $\mathcal{C}_u$ functions in the expected way, and that those functions also preserve $\approx_\beta$.

LEMMA 4.2 (Substitution on kinds).
For all $\Delta, c_k, t, k, \kappa, \kappa_1, u_1, u_2, u$ such that
$\Delta, k{:}u_2 \vdash \kappa_1 : u_1$ and $\Delta, k{:}u_2 \vdash u : \mathsf{Kscm}$ and $\Delta \vdash \kappa : u_2$ and
$k \notin \mathsf{Dom}(c_k)$ and t fresh, then

$$
\begin{array}{lcl}
\mathcal{S}\ [\![\mathsf{Kind}]\!] & = & \Omega \\
\mathcal{S}\ [\![\mathsf{Kscm}]\!] & = & \mathsf{Kind}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}\ c\ [\![X]\!] & = & c(X) \\
\mathcal{C}\ c\ [\![s]\!] & = & \lambda X{:}s.\Pi\_{:}X.\mathcal{S}\ [\![s]\!] \\
\mathcal{C}\ c\ [\![\lambda X{:}A.B]\!] & = & \boldsymbol{\lambda} X{:}A.\boldsymbol{\lambda} Y{:}(\mathcal{C}\ c\ [\![A]\!])X.\mathcal{C}\ \{c, X \mapsto Y\}\ [\![B]\!] \\
\mathcal{C}\ c\ [\![A\ B]\!] & = & (\mathcal{C}\ c\ [\![A]\!])\mathbf{@}B\mathbf{@}(\mathcal{C}\ c\ [\![B]\!]) \\
\mathcal{C}\ c\ [\![\Pi X{:}A.B]\!] & = & \lambda F{:}(\Pi X{:}A.B).\boldsymbol{\Pi} X{:}A.\boldsymbol{\Pi} Y{:}(\mathcal{C}\ c\ [\![A]\!])X. \\
 & & (\mathcal{C}\ \{c, X \mapsto Y\}\ [\![B]\!])(FX)
\end{array}
$$

$A{:}\ B{:}$	$C{:}\ D{:}$	$\boldsymbol{\lambda} X{:}B.C$	$\boldsymbol{\Pi} X{:}B.D$	$C\mathbf{@}A$
Ω	Ω	$\mathsf{fn}\ X{:}B.C$	$B \twoheadrightarrow D$	$C\ A$
Ω	s	C	D	C
s	Ω	$\Lambda^s X{:}B.C$	$\forall_s X{:}B.D$	$C[A]$
s_1	s_2	$\lambda X{:}B.C$	$\Pi X{:}B.D$	$C\ A$

Figure 7. Generalized algorithm on a PTS.

$(\mathcal{C}_k\ \{c_k, k \mapsto t\}\ [\![\kappa_1]\!])[\kappa/k][\mathcal{C}_k\ c_k\ [\![\kappa]\!]/t] \approx_\beta \mathcal{C}_k\ c_k\ [\![\kappa_1[\kappa/k]]\!]$ and $(\mathcal{C}_u\ [\![u]\!])[\kappa/k][\mathcal{C}_k\ c_k\ [\![\kappa]\!]/t] \approx_\beta \mathcal{C}_u\ [\![u[\kappa/k]]\!]$.

LEMMA 4.3 (Substitution on types).
For all $\Delta, c_k, t, \tau, \kappa_1, \kappa_2, u_1, u_2$ such that
$\Delta \vdash \tau : \kappa_1$, and $\Delta, t{:}\kappa_1 \vdash \kappa_2 : u_2$, and $\Delta, t{:}\kappa_1 \vdash u_1 : \mathsf{Kscm}$, and $t \notin \mathsf{Range}(c_k)$, then
$(\mathcal{C}_k\ c_k\ [\![\kappa]\!])[\tau/t] \approx_\beta \mathcal{C}_k\ c_k\ [\![\kappa[\tau/t]]\!]$ and
$(\mathcal{C}_u\ [\![u_1]\!])[\tau/t] \approx_\beta \mathcal{C}_u\ [\![u[\tau/t]]\!]$.

LEMMA 4.4 (Conversion of kinds).
For all $\Delta, \kappa_1, \kappa_2, u_1, u_2$ such that $\Delta \vdash \kappa_1 : u_1$, and $\Delta \vdash \kappa_2 : u_2$, and $\kappa_1 \approx_\beta \kappa_2$, then $\mathcal{C}_k\ c_k\ [\![\kappa_1]\!] \approx_\beta \mathcal{C}_k\ c_k\ [\![\kappa_2]\!]$.

LEMMA 4.5 (Conversion of kind schemas).
For all Δ, u_1, u_2 such that $\Delta \vdash u_1 : \mathsf{Kscm}$, and $\Delta \vdash u_2 : \mathsf{Kscm}$, and $u_1 \approx_\beta u_2$, then $\mathcal{C}_u\ [\![u_1]\!] \approx_\beta \mathcal{C}_u\ [\![u_2]\!]$.

We can most likely also show our translation to be semantics preserving by showing an operational equivalence, i.e. an additional lemma for conversion of types that shows that $\mathcal{C}_t$ also preserves $\approx_\beta$. While we do not foresee any particular problem doing so, we have not considered it yet.

5. Generalization to a Pure Type System

Given the symmetry between $\mathcal{C}_t$, $\mathcal{C}_k$, and $\mathcal{C}_u$, it is tempting to try and rephrase the algorithm within the context of a PTS. Doing it really formally is a bit painful because of the need to distinguish $\mathcal{C}_t$ from the rest since it returns λ_H code rather than TL code. But Fig. 7 shows a slightly sloppy formulation, which closes its eyes on such "details". The algorithm is split into 3 parts:

- First a function $\mathcal{S}$ which needs to be adjusted for each PTS and which documents the hierarchy between the various sorts, as well as where Ω does fit.
- The translation itself, which is really a straightforward adaptation of the algorithm presented in the previous section, except that it uses meta-syntax $\boldsymbol{\lambda}$, $\boldsymbol{\Pi}$, and $\mathbf{@}$ to denote respectively a function term, a function type term, and an application term, where those terms can be either taken from various parts of λ_H or TL.
- A table that shows how to map that meta-syntax to actual syntax, depending on the types of the two subterms.

One could also interpret this algorithm in a pure PTS context, where Ω would simply be an additional sort and the λ_H types and terms would then be replaced by their corresponding PTS terms. I.e. $\boldsymbol{\lambda} X : A.B$ is simply always mapped back to $\lambda X : A.B$ except when that function is dependently typed in which case the argument is dropped to break the dependency.

Rephrasing the algorithm in PTS terms has the advantage of being more concise and pin-pointing more clearly where the differences appear. Also it can help better understand what constraints need to be satisfied by the input and output languages for such a translation to be valid. It seems pretty clear that the destination language needs to be a superset of the input language with one additional sort, which I here called Ω. We have not yet investigated what properties the $\mathcal{S}$ function needs to enjoy, but it appears at least that if the input PTS includes (s_1, s_2) in its $\mathcal{R}$ set, then the output PTS needs to additionally allow $(\mathcal{S}\ [\![s_1]\!], s_2)$, except when $\mathcal{S}\ [\![s_1]\!] = \Omega$.

6. Related work and conclusion

Barthe, Hatcliff, and Sørensen [1999] define CPS translations for pure type systems. They were the main inspiration for the idea of first analyzing the translation algorithm for a stratified presentation of CC, and only afterwards generalize it to pure type systems. They also use similar tricks to our meta-syntax to distinguish between elements in Kind (a.k.a. Prop in their article) and others to recover the stratification made implicit by the use of a PTS.

Minimide, Morrisett, and Harper [1996] develop a typed closure conversion algorithm for a language with intensional type analysis. This is a good example of the kind of trouble you can get into when trying to perform closure conversion on a language that does not enjoy the *phase distinction* property. This should be contrasted to the simple algorithm used in [Morrisett et al., 1998]. While it may be possible to perform closure conversion on CC without going through a transformation like our, that article is a strong indication that it might get ugly.

Crary, Weirich, and Morrisett [2002] show how to reconcile type erasure with intensional type analysis by turning all type arguments into a type argument on the one hand and a corresponding value of singleton type on the other. This recovers the phase distinction property and hence make it possible to use much simpler and more traditional algorithms for compilation steps like closure conversion. We use exactly the same duplication, though for a different purpose: in their case, they want to distinguish uses of types at the level of terms from uses of types at the level of types, whereas we want to distinguish uses of terms at the level of terms from uses of terms at the level of types. Obviously, the same trick would work if we had to deal with both cases (e.g. if our input language was extended with a typecase construct). Their work was an important inspiration for our algorithm.

Crary and Weirich [1999] extend that work by defining a language LX with a richer type language, such that the singleton types can be defined in that language rather than being hard-coded. Their article also presents how to encode inductive types in that language, which could maybe be used to extend this work.

Mishra-Linger and Sheard [2008] propose to extend dependently-typed languages with annotations to indicate which terms can be erased by code extraction, similarly to the Prop-vs-Set distinction in Coq, but via annotations on the function definitions rather than via the type system. This does not solve the problem of how to compile such languages while preserving types, but it could be used as an intermediate language for a code extraction tool.

Fogarty, Pašalić, Siek, and Taha [2007] present an extension of OCaml called Concoqtion which grafts Coq into its type system, in a way similar to what Shao *et al.* suggested with λ_H. Although it is not exactly like λ_H, Concoqtion might work as well as a target language for a type-preserving compilation of a dependently-typed language such as CC.

Hinze [2002] shows how to define polytypic functions indexed by types of arbitrary kinds and these exhibit an uncanny resemblance to our translation function. Of course, this is no accident,

since our translation function $\mathcal{C}_t\ [\![\tau]\!]$ is a polytypic function and it hence inevitably follows the same polykinded pattern as Hinze's functions when we generalize that function to arbitrary kinds and kind schemas.

Discussion Of course, this work is also related to the general issue of combining dependent types and side-effects. Currently, there are fundamentally two approaches to this problem: one is to shun dependent types and rely on GADTs or singleton types instead, as is done in λ_H, the other is to keep dependent types but add monads or some kind of effect system to distinguish pure terms for side effecting terms. The first approach is very popular since it can be added to existing languages and can reuse all the traditional compilation machinery. On the other hand, it forces the user to work much harder to simulate dependent types by duplicating code, thus discouraging the use of dependent types. The other approach on the other hand, makes the use of dependent types much more natural, but has its share of problems as well. One of them is that the jury is still out on what is the best effect or monadic system for it, others are that it is more difficult to manipulate such code while preserving types.

This article shows a way to combine the two approaches. It could be done by using such a wholesale translation like $\mathcal{C}$ in the compilation process, to eliminate the complexity of manipulating dependently typed code. For that we would need to refine the translation so as to distinguish pure from non-pure terms in the input and compile them differently.

Or it could be made more visible directly in the source language. E.g. a language with the phase distinction could simply offer the user the possibility to use types and type functions at the level of terms (maybe transparently or via a special reify request), thus performing the code duplication for them.

This article shows that to some extent λ_H is as expressive as CC, in the sense that its types can express the same constraints as those expressible in CC, but note that this is not *macro-expressible* [Felleisen, 1991] since to apply $\mathcal{C}$ to a term, we may need to apply $\mathcal{C}$ to all the terms to which it refers.

Future work Our plans for the future are to extend the algorithm to inductive definitions with dependent elimination, and to better explore the PTS formulation to try and figure out which requirements need to be satisfied by the source PTS, the destination PTS, and the computation language, for the algorithm to work correctly. Another direction will be to extend the source language to allow side-effects via effect annotations, and see how to adjust the compilation so as to make sure that effectful computations do not end up in the types.

References

Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science (volume 2)*. Oxford Univ. Press, 1991.

G. Barthe, J. Hatcliff, and M.H. Sørensen. CPS-translations and applications: the cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, September 1999.

Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248, Paris, France, September 1999. ACM Press.

Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, December 1991.

Kathleen Fisher, John Reppy, and Jon G. Riecke. A calculus for compiling and linking classes. In *European Symposium on Programming*, volume 1782 of *LNCS*, pages 134–149, New York, NY, March/April 2000. Springer-Verlag.

Seth Fogarty, Emir Pašalić, Jeremy Siek, and Walid Taha. Concoqtion: Indexed types now! In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.

J. Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, University of Paris VII, 1972.

Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, May-June 2002.

Christopher League and Stefan Monnier. Typed compilation against non-manifest base classes. *Lecture Notes in Computer Science*, 3956:77–98, January 2006.

Pierre Letouzey. Extraction in coq: An overview. In *4th conference on Computability in Europe: Logic and Theory of Algorithms*, pages 359–369, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69405-2.

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

Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364, Budapest, Hungary, April 2008.

Stefan Monnier. Typed regions. In *Informal proceedings of the SPACE Workshop*, Venice, Italy, January 2004.

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.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Symposium on Programming Languages Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

John C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

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.

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