The Swiss Coercion

Stefan Monnier
Université de Montréal
monnier@iro.umontreal.ca

Abstract

Recent type systems allow the programmer to use types that describe more precisely the invariants on which the program relies. But in order to satisfy the type system, it often becomes necessary to help the type checker with extra annotations that justify why a piece of code is indeed well-formed. Such annotations take the form of term-level type manipulations, such as type abstractions, type applications, existential package packing and opening, as well as *coercions*, or *casts*. While those operations have no direct runtime cost, they tend to introduce extra runtime operations equivalent to η-redexes or even empty loops in order to get to the point where we can apply that supposedly free operation.

We show a coercion that is like a pacific Swiss army knife of coercions: it cannot cut but it can instantiate, open, pack, abstract, analyze, or do any combination thereof, reducing the need for extra surrounding runtime operations. And all that, of course, for the price of a single coercion, which still costs absolutely nothing at runtime. This new coercion is derived from Karl Crary's coercion calculus [Crary, 2000], but can also replace Crary and Weirich's vcase [Crary and Weirich, 1999].

It additionally happens to come in handy to work around some limitations of value polymorphism. It is presented in the context of Shao et al.'s Type System for Certified Binaries [Shao et al., 2002].

Other than the coercion itself, another contribution of this work is a slightly different proof technique to show soundness of the type erasure.

1. Introduction

Recent type systems allow the programmer to use types that describe more precisely the invariants on which the program relies. This means that types abstract values more closely and that it becomes more frequent to have type mismatches where the two types are although different somehow *compatible*. Also this compatibility is generally too complex to be automatically inferred so in order to satisfy the type system, it often becomes necessary to help the type checker with extra annotations that justify why a piece of code is indeed well-formed. Such annotations take the form of term-level type manipulations, such as *coercions*, or *casts*. While those operations have no direct runtime cost, they tend to introduce extra runtime operations equivalent to η-redexes or even empty loops in order to get to the point where we can apply that supposedly free operation.

Furthermore other type manipulations such as type abstractions, type applications, and existential package packing and opening also become much more frequent, so it is important for the compiler to make those operations efficient, e.g. by enforcing a value restriction on polymorphic quantification. But here again, making those operations free is not always sufficient since they may still require extra runtime operations equivalent to η-redexes.

In [Crary, 2000], Crary presented a coercion calculus that solved this problem by extending coercions such that whole eta-redexes could be written as part of a coercion which can then be implemented as a *no-op*. In our work, using a language similar to Shao et al.'s Type System for Certified Binaries [Shao et al., 2002], we have found this coercion calculus very helpful but have needed to extend it somewhat and also to reformulate it to better fit our language. More specifically we have extended it to fully support polymorphic and existential types. And we have reformulated the coercion calculus itself as an inductive definition in our type language, the calculus of inductive constructions (CIC) [Paulin-Mohring, 1993]. This has the side-benefit of enforcing that our coercions are finite and, more importantly, it allows us to write *dynamic* coercions which depend on type instantiations, i.e. where the precise coercion to apply is computed from the values of some type parameters by a primitive recursive function.

The resulting coercion is like a pacific Swiss army knife of coercions: it cannot cut but it can instantiate, open, pack, abstract, analyze, or do any combination thereof, reducing the need for extra surrounding runtime operations. And all that, of course, for the price of a single coercion, which still costs absolutely nothing at runtime.

Interestingly, by virtue of being expressed within the calculus of inductive constructions, our coercion can be used to implement term-level typecase operations such as Crary and Weirich's vcase [Crary and Weirich, 1999], where the type really does not exist at runtime, so the typecase is really just another kind of *coercion*.

Another bonus is that this coercion helps resolve a particular tension that sometimes appears in heavily typed code: in order for polymorphic quantification to be implemented as a *no-op* the language will typically want to use of some form of value restriction: polymorphic abstraction is only permitted around syntactic values rather than arbitrary code. In SML code, this is a very minor restriction, but in systems that use very refined types, this can become a serious nuisance precisely because polymorphic quantification is needed at many more places. Our coercion is sometimes able to resolve this tension by adding polymorphism after the fact to the result of a computation.

Other than the coercion itself, another contribution of this work is a slightly different proof technique to show soundness of the type erasure: instead of showing first the soundness of a typed semantics, then a type erasure and untyped semantics and finally a bisimulation, we directly show the soundness of the untyped semantics.

Section 2 shows a sample situation that calls for a more powerful coercion. Section 3 presents the necessary background upon which this work builds. Section 4 introduces the language and the coercion itself. Section 5 shows how those coercions can be used. Section 6 formalizes the dynamic semantics and type erasure of the language to prove soundness. Section 7 concludes by presenting related works and open problems left for future work.

2. Motivation

We originally encountered the need for a more powerful form of coercion while working on the *gc_copy* function of a type-preserving stop&copy garbage collector [Monnier et al., 2001]. But related needs appear in many different situations.

GC's copy function The ideal goal here is to write the code for a type preserving deep-copy function that is as close as possible to what we would write in an untyped setting. The idealized untyped version of the code could look something like:

```
(* Copy heap from region F to region T. *)
gc_copy x =
  case x.tag of
    0 ⇒ return x
    1 ⇒ return {tag = 1,
                val = (gc_copy x.val.0, gc_copy x.val.1)^T}
```

Here we intend for x to be a tagged boxed value, where $x.tag$ extracts the tag and $x.val$ extracts the actual value. The value can either be an immediate such an integer or a pointer to a pair in which case we can fetch the two elements using the .0 and .1 selectors. In the case of an immediate value, the copy function does not need to do anything else than return that same tagged immediate value; in the case of a pair, the copy function needs to recursively copy each of the two elements in the pair and then build a new pair (allocated in the destination region T as indicated by the superscript) and tag the resulting pointer before returning it. We will here gleefully ignore issues surrounding low-level representation details and forwarding pointers.

Now if the types of the values we copy only includes integers int and pairs $\tau_1 \times \tau_2$, this can be type checked without too much trouble. But what if your values can also have existential types $\exists t.\tau$ or recursive types $\mu t.\tau$? In λ_H this would mean that before being able to give a type to $x.tag$, we need to open any existential packaging, and unfold any recursion. The problem is: where should we insert those fold/unfold and pack/open operations? Clearly, the only possible place is at the very beginning (and end) of the function just before extracting $x.tag$, but at that point we know nothing about x so we have no idea whether it's indeed a recursively or existentially typed value. Worse: x might have an existential type whose body is itself an existential type, so we may need more than one open/pack before being able to extract $x.tag$.

In current systems this problem can be solved by adding separate tags for values of existential and recursive types. This incurs an overhead which may be acceptable in some cases, but becomes untenable when sophisticated types are used, since these introduce many more uses of existential or universal types.

Now it may seem kind of silly to worry about where to place *no-op*s such as fold/unfold and open/pack operation, in which order and how many of them, since no matter how many of them we introduce they're still *no-op*s. Yet, our type checker needs them.

Repacking in data-structures Another situation which calls for more sophisticated coercions is one where we have a list of integers smaller than 100, whose type could hence be:

$$\mathsf{List}\ (\exists n.\exists P\!:\!n < 100.\mathsf{snat}\ n)$$

(*sort*)	s	$::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext}$	
(*ptm*)	τ, κ, φ, P	$::= s \mid x \mid \lambda x\!:\!\varphi.\,\varphi \mid \varphi\ \varphi \mid \Pi x\!:\!\varphi.\,\varphi$ $\mid \mathsf{Ind}(x\!:\!\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i, \varphi) \mid \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\}$	
(*funcs*)	f	$::= \lambda x\!:\!\tau \Rightarrow e$	Function
		$\mid \Lambda t\!:\!\kappa.f$	Type abstraction
(*terms*)	e	$::= n$	Integer
		$\mid x$	Variable
		$\mid f$	Polymorphic function
		$\mid e_1\ e_2$	Apply e_1 to e_2
		$\mid e[\varphi]$	Type application
		$\mid \langle t\!=\!\varphi_1, e\!:\!\varphi_2\rangle$	Existential package
		$\mid \mathsf{let}\ \langle t, x\rangle = e_1\ \mathsf{in}\ e_2$	Open e_1 in e_2
		$\mid (e_0, ..., e_{n-1})$	Tuple of size n
		$\mid \pi[P]\ e_1\ e_2$	Project field e_1 of tuple e_2

Figure 1. Syntax of the λ_H language.

where snat n denotes the singleton type of the constant n. If we want to pass this list to a function that expects a list of integers smaller than 200, we need to weaken the type to:

$$\mathsf{List}\ (\exists n.\exists P\!:\!n < 200.\mathsf{snat}\ n)$$

Weakening any one of those integers from $\exists n.\exists P\!:\!n < 100.\mathsf{snat}\ n$ to $\exists n.\exists P\!:\!n < 200.\mathsf{snat}\ n$ is just a matter of unpacking and repacking existentials and can be done at no cost, but doing it on the whole list would typically require looping over the list in order to apply the *no-op* on each element. The coercion calculus of [Crary, 2000] was specifically intended to eliminate such runtime cost by making it possible to do the loop directly within the coercion. But this does not help us here, because Crary's coercion calculus does not allow this kind of unpacking and repacking, except in the rare case where the witness is not used[1].

3. Background

We introduce in this section the two main works on which we build this article: the λ_H language presented in [Shao et al., 2002] to which we add a coercion primitive; and the coercion calculus presented in [Crary, 2000] which we rephrase and extend by presenting it in the context of λ_H.

3.1 λ_H

In [Shao et al., 2002], Shao et al. presented a programming language λ_H that can represent and manipulate arbitrarily complex propositions and proofs, by using CiC as their type language. More specifically, λ_H types are CiC terms, and hence CiC types are λ_H kinds. Figure 1 shows part of the syntax of the language in BNF notation. The first two syntactic categories of sorts s and pure type terms τ, κ, φ, P is just a concise representation of the syntax of CiC terms. The term language is a polymorphic λ-calculus, where the polymorphism enforces the value restriction by allowing type abstraction only on functions (via the syntactic category f); an existential package $\langle t = \varphi_1, e : \varphi_2\rangle$ packs e with the witness φ_1 to create an object of type $\exists t.\varphi_2$ and can be opened with the let form; the language also includes n-tuples and has the particularity that the field index of the projection function π can be an arbitrary expression rather than just a constant, so tuples can also be used as arrays; to guarantee the soundness of the language, the field access operation requires an additional type annotation P which proves that the index is within bounds.

[1] To be fair, Crary's coercion calculus does not include existential types, so this limitation really only applies to the obvious generalization of his work to existential types.

The figure does not show the available types: in a traditional system, we would add a separate category for types which could look like the following:

$$(\textit{types})\ \tau ::= t \mid \mathsf{snat}\ n \mid \mathsf{tup}\ n\ \varphi \mid \tau \to \tau \mid \forall t{:}\kappa.\tau \mid \exists t{:}\kappa.\tau$$

Where $\mathsf{snat}\ n$ is the type used for natural numbers; $\mathsf{tup}\ n\ \varphi$ is the type used for tuples of size n; and the last three are the usual function types, polymorphic types, and existential types. But this would be deceptive here: the type language is much richer than that. Instead, these type constructors are created directly within CiC as part of the inductive definition of the Ω kind[2]:

$$\begin{array}{l}\mathsf{Inductive}\ \Omega\ :\ \mathsf{Kind}\ :=\\ \quad \mid \mathsf{snat}\colon\ \mathit{nat} \to \Omega\\ \quad \mid \mathsf{tup}\ :\ \mathit{nat} \to (\mathit{nat} \to \Omega) \to \Omega\\ \quad \mid \mathsf{arw}\ :\ \Omega \to \Omega \to \Omega\\ \quad \mid \forall\ \ :\ (k \to \Omega) \to \Omega\\ \quad \mid \exists\ \ :\ (k \to \Omega) \to \Omega\end{array}$$

The type $\mathsf{snat}\ n$ is the singleton type of the natural constant n; the type $\mathsf{tup}\ n\ \varphi$ is the type of tuples of size n where φ is a type-level function that takes a field index as argument and returns its type; the type $\mathsf{arw}\ \tau_1\ \tau_2$ is the usual type of functions that take arguments of type τ_1 and return values of type τ_2; the type $\forall\ \varphi$ is the polymorphic type traditionally written $\forall t : \kappa.\varphi\ t$ and similarly the type $\exists\ \varphi$ is the existential type traditionally written $\exists t : \kappa.\varphi\ t$: we encode the usual quantifier types using a form of higher-order abstract syntax [Pfenning and Elliott, 1988].

The λ_H language is not dependently typed: it keeps the usual phase distinction between types and terms, and hence type checking is decidable even in the presence of side-effects and non termination. As a matter of fact, the term language and type language are very independent, since the only way in which they are linked is basically by using CiC as the language in which to write types, which mostly amounts to defining the inductive Ω kind. So although the language presented in [Shao et al., 2002] is a kind of polymorphic λ-calculus, the same underlying idea can be applied just as easily to other languages, as was done in [Monnier, 2004].

Although λ_H is not dependently typed, the reasoning power provided by the language is comparable to that of dependently typed languages. This is obtained by using singleton types: for example, the type $\mathsf{snat}\ 3$ has only one member, which is the natural number 3. This amounts to lifting *values* to the level of types and hence provides the limited form of dependency necessary to enable powerful reasoning: e.g. [League and Monnier, 2006] uses a variant of λ_H to express and verify the invariants used in an encoding of sophisticated object oriented features such as non-manifest base classes.

3.2 The calculus of coercions

As part of his work on compiling away subtyping (using bounded polymorphism) down to intersection types [Crary, 2000], Crary introduced a coercion calculus which can apply coercions not only on simple values but also inside data-structures and on functions, removing the common need to keep η-redexes and empty loops at runtime just for the sake of executing some coercion. The coercion themselves follow the following syntax:

$$c ::= \mathsf{id} \mid c_1 \circ c_2 \mid c_1 \to c_2 \mid c_1 \times c_2 \mid \mathsf{app}\ \tau \mid \forall t.c \mid \mathsf{gen}$$

id is the trivial identity coercion; $c_1 \circ c_2$ is the composition of two coercions; $c_1 \to c_2$ is the congruence coercion for functions which applies the coercion c_1 to the function's argument and c_2 to the function's return result; $c_1 \times c_2$ is the congruence coercion on pairs which applies the coercion c_1 to the first element of a pair and c_2 to the second; $\mathsf{app}\ \tau$ is the primitive coercion which eliminates polymorphism by doing a type application; finally $\forall t.c$ is the congruence coercion on polymorphic values which applies the coercion c to the return value of the type abstraction; and gen is the primitive coercion which introduces polymorphism by wrapping a value inside a trivial type abstraction.

[2] To make the notation more lightweight, we use a convention similar to that used by Twelf whereby most of the Π quantifiers are left implicit

$$\boxed{\Delta \vdash c : \tau_1 \lhd \tau_2 \quad \mid \quad c \text{ is a valid coercion from } \tau_1 \text{ to } \tau_2}$$

$$\frac{}{\Delta \vdash \mathsf{id} : \tau \lhd \tau} \qquad \frac{\Delta \vdash c_1 : \tau_1 \lhd \tau_2 \qquad \Delta \vdash c_2 : \tau_2 \lhd \tau_3}{\Delta \vdash c_1 \circ c_2 : \tau_1 \lhd \tau_3}$$

$$\frac{\Delta \vdash c_1 : \tau_{21} \lhd \tau_{11} \qquad \Delta \vdash c_2 : \tau_{12} \lhd \tau_{22}}{\Delta \vdash c_1 \to c_2 : \tau_{11} \to \tau_{12} \lhd \tau_{21} \to \tau_{22}}$$

$$\frac{\Delta \vdash c_1 : \tau_{11} \lhd \tau_{21} \qquad \Delta \vdash c_2 : \tau_{12} \lhd \tau_{22}}{\Delta \vdash c_1 \times c_2 : \tau_{11} \times \tau_{12} \lhd \tau_{21} \times \tau_{22}}$$

$$\frac{}{\Delta \vdash \mathsf{app}\ \tau_2 : \forall t.\tau_1 \lhd \tau_1[\tau_2/t]} \qquad \frac{\Delta, t \vdash c : \tau_1 \lhd \tau_2}{\Delta \vdash \forall t.c : \forall t.\tau_1 \lhd \forall t.\tau_2}$$

$$\frac{t \notin \mathit{fv}(\tau)}{\Delta \vdash \mathsf{gen} : \tau \lhd \forall t.\tau}$$

Figure 2. Formation rules for Crary's coercions

Clearly, since coercions include type application as a special case, Crary's term language does not need to provide a separate type application construct. Coercions are even able to do type application inside data-structures and functions without opening those data-structures or calling those functions.

Coercions are comparable to witnesses of subtyping. They have types of the form $\tau_1 \lhd \tau_2$ which means that a term of type τ_1 can be coerced to type τ_2, or equivalently that τ_1 is a subtype of τ_2. Figure 2 shows the formation rules. The identity and composition rules are straightforward, as is the congruence on pairs; the congruence on functions obeys of course the contra-variance rule; the type application rule just reproduces the normal typing rule of type application; finally the congruence on polymorphic types needs to adjust the type environment which is used elsewhere (not shown here) to ensure that the types are themselves well-formed; and the type abstraction rule is restricted to only allow introduction of polymorphism in the case where the quantified variable is actually not used. This last restriction is the main one we will want to lift in this article.

Crary's coercion calculus also includes additional coercions to introduce and eliminate intersection types as well as a top type which is the supertype of all types. It did not include existential types, but since they are dual to polymorphic types, they can be added easily. Another important element that was included, on the other hand, was recursive types which introduced a significant amount of complexity, since they came with 4 coercions: fold and unfold of course, but also rec and isorec which provided 2 different forms of congruence on recursive types, one where the induction hypothesis was always covariant with the conclusion, and the other where both covariant and contravariant hypothesis were available.

4. The language $\lambda_{\mathbf{CH}}$

In this section we will formally present the language's syntax, typing rules, and dynamic semantics.

$$
\begin{array}{llll}
(\textit{sort}) & s & ::= & \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{ptm})\,\tau, \kappa, \varphi, P & & ::= & s \mid x \mid \lambda x\!:\!\varphi.\,\varphi \mid \varphi\ \varphi \mid \Pi x\!:\!\varphi.\,\varphi \\
 & & & \mid \mathsf{Ind}(x\!:\!\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i, \varphi) \mid \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\} \\
(\textit{funcs}) & f & ::= & \lambda x\!:\!\tau \Rightarrow e \qquad \text{Function} \\
 & & & \mid \Lambda t\!:\!\kappa.f \qquad \text{Type abstraction} \\
(\textit{terms}) & e & ::= & n \qquad \text{Integer} \\
 & & & \mid x \qquad \text{Variable} \\
 & & & \mid f \qquad \text{Polymorphic function} \\
 & & & \mid e_1\ e_2 \qquad \text{Apply } e_1 \text{ to } e_2 \\
 & & & \mid \mathsf{let}\ \langle t, x\rangle = e_1\ \mathsf{in}\ e_2 \qquad \text{Open } e_1 \text{ in } e_2 \\
 & & & \mid (e_0, ..., e_{n-1}) \qquad \text{Tuple of size } n \\
 & & & \mid \pi[P]\ e_1\ e_2 \qquad \text{Project field } e_1 \text{ of tuple } e_2 \\
 & & & \mid \mathsf{cast}[P]\ e \qquad \text{Apply coercion } P \text{ to } e
\end{array}
$$

Figure 3. Syntax of λ_{CH}.

4.1 Syntax of λ_{CH}

Figure 3 shows the syntax of the language λ_{CH}, which is basically a variant of λ_H [Shao et al., 2002] extended with our Swiss coercions. It is a typed λ-calculus with natural constants (n), functions ($\lambda x\!:\!\tau \Rightarrow e$), applications ($e_1\ e_2$), tuples $(e_0, ..., e_{n-1})$, polymorphic functions ($\Lambda t\!:\!\kappa.f$), as well as existential types, opened via the form $\mathsf{let}\ \langle t, x\rangle = e_1\ \mathsf{in}\ e_2$. In order for the type applications to be no-ops at runtime, we apply the *value restriction* by imposing that only *functions* can be polymorphic. For access to tuple fields, instead of the usual fixed-field selection ($\pi_i\ e$), we use a more general form ($\pi[P]\ e_1\ e_2$) where the field index is an arbitrary expression, so tuples can also be used as arrays; for this reason the form takes an additional argument P which is a proof that the index is indeed within bounds. Finally, the language includes coercion expressions ($\mathsf{cast}[P]\ e$) where P describes the coercion to apply. Note that the language includes neither type application expressions nor expressions to create existential packages: instead, those are provided indirectly as special cases of $\mathsf{cast}[P]\ e$ expressions.

Following the approach used for λ_H, the type (and kind) language is the Calculus of Inductive Constructions (CiC) [Paulin-Mohring, 1993]. In such a system, instead of including the traditional type constructors directly in the type language, they are defined *within* the type language as one particular inductive kind:

$$
\begin{array}{l}
\mathsf{Inductive}\ \Omega\ :\ \mathsf{Kind}\ := \\
\quad \mid \mathsf{snat}\colon\ \mathit{nat} \to \Omega \\
\quad \mid \mathsf{tup}\ :\ \mathit{nat} \to (\mathit{nat} \to \Omega) \to \Omega \\
\quad \mid \mathsf{arw}\ :\ \Omega \to \Omega \to \Omega \\
\quad \mid \forall\ \ :\ (k \to \Omega) \to \Omega \\
\quad \mid \exists\ \ :\ (k \to \Omega) \to \Omega \\
\quad \mid \mu\ \ :\ (k \to k) \to (k \to \Omega) \to \Omega
\end{array}
$$

The type $\mathsf{snat}\ \ n$ is the singleton type of the natural constant n: every constant has its own type. The type $\mathsf{tup}\ n\ \varphi$ is the type of tuples of size n, φ is a function that describes the type of each field: it takes the index of a field and returns its type. The type $\mathsf{arw}\ \tau_1\ \tau_2$ is the type of functions of argument type τ_1 and return type τ_2. The types $\forall\ \varphi$ and $\exists\ \varphi$ are the universal and existential types, respectively, where the quantification can be over types of any kind. We will often use the more common notation $\forall t : \kappa.\tau$ to stand for $\forall\ (\lambda t : \kappa.\tau)$, and similarly for $\exists$. Finally the type $\mu\ \varphi_1\ \varphi_2$ is the recursive type which stands for the infinite expansion $\varphi_2(\varphi_1(\varphi_1(\varphi_1(...(\bot)))))$. Although this representation allows recursive types of higher kinds, we will only use it here with kind Ω, in which case φ_2 can be forced to the identity function[3].

[3] See [Collins and Shao, 2002] for how to use such a construct with higher kinds.

$\Delta \vdash \Gamma$	Γ is well-formed in type environment Δ
$\Delta; \Gamma \vdash e : \tau$	e has type τ in environments Δ and Γ

$$\frac{}{\Delta \vdash \bullet} \qquad \frac{\Delta \models^{\text{CIC}} \tau : \Omega \qquad \Delta \vdash \Gamma}{\Delta \vdash \Gamma, x\!:\!\tau}$$

$$\frac{}{\Delta; \Gamma \vdash n : \mathsf{snat}\ n} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta \models^{\text{CIC}} \tau_1 : \Omega \qquad \Delta; \Gamma, x\!:\!\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x\!:\!\tau_1 \Rightarrow e : \mathsf{arw}\ \tau_1\ \tau_2}\ \textit{(I-arw)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{arw}\ \tau_1\ \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}\ \textit{(E-arw)}$$

$$\frac{\Delta \models^{\text{CIC}} \kappa : \mathsf{Kind} \qquad \Delta, t\!:\!\kappa; \Gamma \vdash f : \tau}{\Delta; \Gamma \vdash \Lambda t\!:\!\kappa.f : \forall t\!:\!\kappa.\tau}\ \textit{(I-}\forall\textit{)}$$

$$\frac{\begin{array}{c}\Delta, t\!:\!\kappa; \Gamma, x\!:\!\tau_1 \vdash e_2 : \tau_2 \\ \Delta; \Gamma \vdash e_1 : \exists t\!:\!\kappa.\tau_1 \qquad \Delta \models^{\text{CIC}} \tau_2 : \Omega\end{array}}{\Delta; \Gamma \vdash \mathsf{let}\ \langle t, x\rangle = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textit{(E-}\exists\textit{)}$$

$$\frac{\forall i \quad \Delta; \Gamma \vdash e_i : \varphi\ i}{\Delta; \Gamma \vdash (e_0, ..., e_{n-1}) : \mathsf{tup}\ n\ \varphi}\ \textit{(I-tup)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_2 : \mathsf{tup}\ \varphi_2\ \varphi \\ \Delta; \Gamma \vdash e_1 : \mathsf{snat}\ \varphi_1 \qquad \Delta \models^{\text{CIC}} P : \varphi_1 \le \varphi_2\end{array}}{\Delta; \Gamma \vdash \pi[P]\ e_1\ e_2 : \varphi\ \varphi_1}\ \textit{(E-tup)}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \qquad \Delta \models^{\text{CIC}} P : \tau_1 \lhd \tau_2}{\Delta; \Gamma \vdash \mathsf{cast}[P]\ e : \tau_2}\ \textit{(Cast)}$$

Figure 4. Typing rules of λ_{CH}.

We will also use the more common notation $\mu t.\tau$ to stand for $\mu\ (\lambda t.\tau)\ (\lambda t.t)$.

4.2 Static semantics of λ_{CH}

Figure 4 shows the static semantics of λ_{CH}. The constant and variable rules are as uneventful as they should be, except for the fact that natural numbers have singleton types and that we abuse the notation, writing n both at term and type levels, but really these are two different terms representing the same number in two different ways: once as a value and once as a type. The introduction and elimination rules for arw, $\forall$, and $\exists$ are equally straightforward, the only unusual part being the recourse to the typing rule of CiC ($\models^{\text{CIC}}$) to check well-formedness of types and kinds, as well as to check that the witness type does not escape from the let package opening construct. The I-tup rule checks that the tuple has the same number of elements as specified in its type and that φ properly describes the type of each element. The E-tup rule checks that the index is indeed a number and that the object from which to select is indeed a tuple and it additionally uses CiC to check that P is a proof that the index is within bounds. Finally the *Cast* rule implements the coercion itself by checking the proof P which specifies the coercion to perform.

The valid coercions are determined by a subtyping relation $\lhd$ which is also defined within the type language as another inductive

$$
\begin{array}{lll}
(\textit{funcs})\ \bar{f} ::= \lambda x \Rightarrow \bar{e} & & \text{Function} \\
(\textit{terms})\ \bar{e} ::= n & & \text{Integer} \\
\quad \mid x & & \text{Variable} \\
\quad \mid \bar{f} & & \text{Function} \\
\quad \mid (\bar{e}_0, ..., \bar{e}_{n-1}) & & \text{Tuple of size } n \\
\quad \mid \bar{e}_1\ \bar{e}_2 & & \text{Apply } \bar{e}_1 \text{ to } \bar{e}_2 \\
\quad \mid \pi\ \bar{e}_1\ \bar{e}_2 & & \text{Project field } \bar{e}_1 \text{ of tuple } \bar{e}_2
\end{array}
$$

Figure 5. Syntax of λ_{CU}.

kind:

$$
\begin{array}{l}
\mathsf{Inductive}\ \lhd\ :\ \Omega \to \Omega \to \mathsf{Kind}\ := \\
\quad \mid \mathsf{cid}\ \ : t \lhd t \\
\quad \mid \mathsf{ccomp}\ : t_1 \lhd t_2 \to t_2 \lhd t_3 \to t_1 \lhd t_3 \\
\quad \mid \mathsf{ctup}\ \ : (\Pi\, i, i < n \to \varphi_1\ i \lhd \varphi_2\ i) \to \mathsf{tup}\ n\ \varphi_1 \lhd \mathsf{tup}\ n\ \varphi_2 \\
\quad \mid \mathsf{carw}\ \ : t_{d2} \lhd t_{d1} \to t_{r1} \lhd t_{r2} \to \mathsf{arw}\ t_{d1}\ t_{r1} \lhd \mathsf{arw}\ t_{d2}\ t_{r2} \\
\quad \mid \mathsf{cpack}\ : \varphi\ t \lhd \exists\ \varphi \\
\quad \mid \mathsf{ctapp}\ : \forall\ \varphi \lhd \varphi\ t \\
\quad \mid \mathsf{copen}\ \ : (\Pi\, t, \varphi\ t \lhd t_2) \to \exists\ \varphi \lhd t_2 \\
\quad \mid \mathsf{cgen}\ \ : (\Pi\, t, t_1 \lhd \varphi\ t) \to t_1 \lhd \forall\ \varphi \\
\quad \mid \mathsf{cfold}\ \ : \varphi_1(\mu\ \varphi_1\ \mathsf{id}) \lhd \mu\ \varphi_1\ \mathsf{id} \\
\quad \mid \mathsf{cunfold}\colon \mu\ \varphi_1\ \mathsf{id} \lhd \varphi_1(\mu\ \varphi_1\ \mathsf{id})
\end{array}
$$

This inductive definition defines both the syntax and the formation rules of coercions and should thus be compared to the formation rules of Crary's coercions: cid, ccomp, carw, and ctapp correspond directly to the formation rules of Crary's id, $c_1 \circ c_2$, $c_1 \to c_2$, and app τ respectively; ctup is a natural extension to the case of n-tuples of Crary's rule for $c_1 \times c_2$; cpack is the natural dual of ctapp; cfold and cunfold also match Crary's rules for his fold and unfold coercions; the difference is mostly in the introduction of polymorphism: where Crary had basically:

$$
\begin{array}{l}
\quad \mid \mathsf{cpoly}\colon\ (\Pi\, t, \varphi_1\ t \lhd \varphi_2\ t) \to \forall\ \varphi_1 \lhd \forall\ \varphi_2 \\
\quad \mid \mathsf{cgen'}\colon\ \varphi \lhd \forall\ (\lambda t.\varphi)
\end{array}
$$

We use instead the single cgen rule:

$$
\quad \mid \mathsf{cgen}\colon\ (\Pi\, t, t_1 \lhd \varphi\ t) \to t_1 \lhd \forall\ \varphi
$$

This subsumes the two other rules: cgen' is a combination of cid and cgen while cpoly can be built by judicious use of cgen and ctapp. copen is of course just the dual of cgen. Note that the two rules cpack and ctapp can also be written:

$$
\begin{array}{l}
\quad \mid \mathsf{cpack'}\colon\ \exists\ \varphi \lhd t_2 \to (\Pi\, t, \varphi\ t \lhd t_2) \\
\quad \mid \mathsf{ctapp'}\colon\ t_1 \lhd \forall\ \varphi \to (\Pi\, t, t_1 \lhd \varphi\ t)
\end{array}
$$

Which is a bit more verbose, but has the advantage of making them more obviously inverses of copen and cgen respectively.

4.3 Dynamic semantics of λ_{CH}

The dynamic semantics of our language λ_{CH} is defined indirectly by first compiling the source code into the untyped language λ_{CU}. Figure 5 shows the syntax of λ_{CU}, which is identical to λ_{CH} except that every type annotation has disappeared and type related operations have also been removed: type abstractions, let constructs to open existential packages, and the coercions.

Figure 6 shows the translation from λ_{CH} terms to λ_{CU} terms, which is just a type erasure: every term's type annotations are simply dropped. Type abstractions are replaced with their bodies, coercions are similarly eliminated. The only slightly less trivial translation is for the let construct which in λ_{CH} opens up existential packages, but in λ_{CU} does not exist, and is translated into a standard encoding of the let binding using a trivial combination of function abstraction and function application.

$$
\begin{array}{ll}
\ulcorner n \urcorner & = n \\
\ulcorner x \urcorner & = x \\
\ulcorner \lambda x\!:\!\tau \Rightarrow e \urcorner & = \lambda x \Rightarrow \ulcorner e \urcorner \\
\ulcorner \Lambda t\!:\!\kappa.f \urcorner & = \ulcorner f \urcorner \\
\ulcorner (e_0, ..., e_i) \urcorner & = (\ulcorner e_0 \urcorner, ..., \ulcorner e_i \urcorner) \\
\ulcorner \pi[P]\ e_1\ e_2 \urcorner & = \pi\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner \\
\ulcorner e_1\ e_2 \urcorner & = \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner \\
\ulcorner \mathsf{let}\ \langle t, x \rangle = e_1\ \mathsf{in}\ e_2 \urcorner & = (\lambda x \Rightarrow \ulcorner e_2 \urcorner)\ \ulcorner e_1 \urcorner \\
\ulcorner \mathsf{cast}[P]\ e \urcorner & = \ulcorner e \urcorner
\end{array}
$$

Figure 6. Type erasure

$\bar{e} \rightsquigarrow \bar{e}'$ | $\bar{e}$ reduces in one step to $\bar{e}'$

$$
\frac{i < n}{\pi\ i\ (\bar{e}_0, ..., \bar{e}_{n-1}) \rightsquigarrow \bar{e}_i} \qquad \frac{}{(\lambda x \Rightarrow \bar{e}_1)\ \bar{e}_2 \rightsquigarrow \bar{e}_1[\bar{e}_2/_x]}
$$

$$
\frac{\bar{e}_i \rightsquigarrow \bar{e}'_i}{(\bar{e}_0, ..., \bar{e}_i, ..., \bar{e}_{n-1}) \rightsquigarrow (\bar{e}_0, ..., \bar{e}'_i, ..., \bar{e}_{n-1})}
$$

$$
\frac{e_1 \rightsquigarrow e'_1}{\pi\ e_1\ e_2 \rightsquigarrow \pi\ e'_1\ e_2} \qquad \frac{e_2 \rightsquigarrow e'_2}{\pi\ e_1\ e_2 \rightsquigarrow \pi\ e_1\ e'_2} \qquad \frac{e_1 \rightsquigarrow e'_1}{e_1\ e_2 \rightsquigarrow e'_1\ e_2}
$$

$$
\frac{e_2 \rightsquigarrow e'_2}{e_1\ e_2 \rightsquigarrow e_1\ e'_2}
$$

Figure 7. Operational semantics of λ_{CU}.

The dynamic semantics of λ_{CU} is shown in Fig. 7 as a small step reduction relation $\rightsquigarrow$. It is a bog-standard semantics of λ-calculus with tuples. The first two rules are the primitive reduction steps while the rest are congruence rules. The reduction ordering is non-deterministic, mostly because it made the presentation marginally simpler, but also because it implies that the formal properties shown in Sec. 6 are valid for both call-by-value and call-by-name strategies.

5. Examples

In this section, we will show how to solve some particular problems in our language. More specifically, we will show how we can do the equivalent of Crary and Weirich's vcase, and even extend it in various ways; and then we will show how our coercion can work around some common limitations due to the value restriction.

5.1 Mimicking vcase

In [Crary and Weirich, 1999], Crary and Weirich present a language LX that is able to perform intensional type analysis, without it being directly provided by the language. The way it works is that the language provides a powerful type language with primitive recursion and singleton types and then they cleverly encode runtime type representations as singleton types accompanied by proofs of impossibility (i.e. objects of type void which is not inhabited). They then use those impossibility proofs together with a special term-level *typecase* construct called vcase to reflect the runtime information recovered by plain case statements back into the type.

This vcase construct is very unusual: it is a construct at the term-level which decides the control-flow based solely on type information not available at runtime. The twist to make this impossibility possible is that the typing rule of vcase enforces that all but one of the branches immediately return a value of type void, which implies

that only one branch will ever be taken since there are no values of type void.

The construct looks as follows:

$$\begin{array}{l}\mathsf{vcase}\ \tau\ \mathsf{of}\ \mathit{injl}\ t \Rightarrow \mathsf{dead}\ v\\ \qquad\qquad \mathit{injr}\ t \Rightarrow e\end{array}$$

Here we can see that all but one branch are marked as dead, and the argument to dead has to be a *value* of type void. Of course, for this to be usable, the type of v is not just plain void but rather some type expression which reduces to void in the case that $\tau = \mathit{injl}\ t$. The interest of this construct is that it does *type refinement*, so within e the type environment reflects the fact that τ is in fact equal to $\mathit{injr}\ t$. So this can be used to reflect into the type system information obtained at runtime.

Our language λ_{CH} is at least as powerful as LX in terms of its ability to use type-level primitive recursion as well as singleton types, so the same kind of programming tricks can be used, and indeed the *gc_copy* function in Sec. 2 is in the need to use a form of runtime type analysis. Luckily, our Swiss coercion coupled with CiC's built-in case analysis and primitive recursion allows us to encode the same kind of construct as vcase as follows:

$$\begin{array}{l}\mathit{goal} = \forall\ (\lambda t.\varphi\ (\mathit{injr}\ t)) \lhd \varphi\ \tau\\ \mathit{proof} = \mathsf{case}\ \tau\ \mathsf{of}\ \mathit{injl}\ t \Rightarrow \varphi_{\mathsf{imp}}\ \mathit{goal}\\ \qquad\qquad\qquad\qquad \mathit{injr}\ t \Rightarrow \mathsf{ctapp}\\ \mathsf{cast}[\mathit{proof}]\Lambda t.e\end{array}$$

In the LX code, we had that e was typed in an environment augmented with t and knowledge that $\tau = \mathit{injr}\ t$, and the other branch was ruled out by the type of v. The λ_{CH} code is similar except that the type refinement is not as sophisticated; it is nevertheless sufficient since we can choose φ to be a type-level function of the form $\lambda t'.\forall P : t' = \tau \ldots$: before the cast we cannot easily instantiate P, but after the cast, its kind has turned into $\tau = \tau$ so P can be instantiated trivially.

Another important difference is that in LX, the impossibility proof is provided by a value of type void whereas in our encoding, this is provided by a type expression φ_{imp} which should have kind False (a non-inhabited kind). The difference is not very important in practice since those impossible proofs do not really exist anyway: they either accompany the real data as an extra tuple slot in LX or they accompany it as an existential package witness in λ_{CH}. Actually, storing this "data" in existential witnesses rather than in tuple fields is preferable since it makes it clear that it has no runtime cost.

Finally, the most significant difference is that the vcase construct is weird and ad-hoc, whereas our encoding in λ_{CH} only uses common constructs without arbitrary restrictions. For example, we can easily accommodate the situation where more than one branch in the case is possible (as long as all the branches execute the exact same code), or where one of the branch needs to "do nothing and then recurse".

This last situation is exactly the one we faced in the *gc_copy* function in Sec. 2: we need a coercion at the toplevel of the function which checks the type of the argument: if it is a recursive type, we need to unfold it and then try again with the result, if it is an existential type we similarly need to open it and try again with its content, and this repeated until we hit a real data type such as int or $\tau_1 \times \tau_2$. Notice how all three branches of the corresponding vcase are perfectly possible, but that after some unspecified number of *no-op*s the control-flow will eventually end up in the third branch. We also of course trivially know that the number of *no-op*s will not be infinite, since those *no-op*s are specified as a CiC inductive type and CiC is strongly normalizing.

$$\begin{array}{lll}(\mathit{values})\ \bar{v} ::= n & & \text{Integer}\\ \quad \mid (\bar{v}_0, \ldots, \bar{v}_{n-1}) & & \text{Tuple of size } n\\ \quad \mid \lambda x \Rightarrow \bar{e} & & \text{Function}\end{array}$$

Figure 9. Values

5.2 Defeating the value restriction

The value restriction is a constraint that ensures that type abstraction is only ever applied to values. There are a few different variants of this concept such as SML's restriction to ensure sound type inference [Wright, 1995], or Haskell's restriction to avoid pathological inefficiencies. The variant that interests us here is the stricter one which is concerned with making sure that a type application $v[\varphi]$ can be compiled into a *no-op*s. This is crucial to be able to compile highly polymorphic code efficiently, as well as to be able to implement closure conversion sanely [Minimide et al., 1996, Morrisett et al., 1998].

But this restriction is sometimes very inconvenient. For example, it prevents you from writing code such as:

$$\Lambda t.(f_1[t], f_2[t], f_3[t])$$

This kind of construction can occur for example when building dictionaries in the implementation of object-oriented features [League et al., 2002, League and Monnier, 2006]. But this is no match to our coercion since we can now first construct the dictionary without instantiating each function and then use the Swiss coercion to introduce the outer type abstraction and to instantiate the inner polymorphic elements.

6. Soundness

Rather than show the soundness of the type-erased code by first defining a sound typed semantics and then showing a bisimulation between the typed semantics and the untyped semantics, we prove soundness of the untyped semantics directly. To this end, we first define a type system for λ_{CU}, then show that type erasure of a properly typed λ_{CH} program results in a properly typed λ_{CU} program, and finally show the soundness of λ_{CU}.

Figure 8 shows the static semantics of λ_{CU}. The rules are almost one-to-one equivalent to the rules of λ_{CH} except for minor differences in the *E*-$\exists$ rules. Basically, λ_{CU} is the Curry-style presentation of the language, whereas λ_{CH} uses the Church-style presentation. Notice how there are two rules that can apply to an application $e_1\ e_2$: either the *E-arw* or the *E*-$\exists$, this is due to the fact that the let construct of λ_{CH} which opens existential packages is type-erased into an application.

THEOREM 6.1 (Type preserving erasure).
If $\Delta; \Gamma \vdash e : \tau$, *then* $\Delta; \Gamma \vdash \ulcorner e \urcorner : \tau$.
The proof is a straightforward induction over the typing derivation.

Now that we know that the type-erasure of a well-typed program in λ_{CH} is a well-typed program in λ_{CU}, we only have to show that λ_{CU} is sound. To do that we first need to distinguish between a stuck state and a valid final state. To this end we define in Fig. 9 the subset of expressions of λ_{CU} that are considered as *values*; these include constant natural numbers, tuples made up of values and λ abstractions.

The rest of the proof follows the usual substitution, canonical forms, progress, and preservation arguments. A few things to note before we start: typing a program in λ_{CU} is in general not decidable; luckily we do not need to do that: we only need to be able to check and manipulate typing derivations which already give us the typing of the program. I.e. we basically use the annotations in λ_{CH} to guide the type checking of the type erased program.

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

$$\frac{\Gamma(x) = \tau}{\Delta;\Gamma \vdash x : \tau} \qquad \frac{}{\Delta;\Gamma \vdash n : \mathsf{snat}\ n}\ \textit{(I-snat)} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1 \qquad \Delta \vdash^{\text{CIC}} P : \tau_1 \lhd \tau_2}{\Delta;\Gamma \vdash e : \tau_2}\ \textit{(Cast)}$$

$$\frac{\forall i \quad \Delta;\Gamma \vdash \bar{e}_i : \varphi\ i}{\Delta;\Gamma \vdash (\bar{e}_0, ..., \bar{e}_{n-1}) : \mathsf{tup}\ n\ \varphi}\ \textit{(I-tup)} \qquad \frac{\Delta;\Gamma \vdash \bar{e}_1 : \mathsf{snat}\ \varphi_1 \qquad \Delta;\Gamma \vdash \bar{e}_2 : \mathsf{tup}\ \varphi_2\ \varphi \qquad \Delta \vdash^{\text{CIC}} P : \varphi_1 \leq \varphi_2}{\Delta;\Gamma \vdash \pi\ \bar{e}_1\ \bar{e}_2 : \varphi\ \varphi_1}\ \textit{(E-tup)}$$

$$\frac{\Delta \vdash^{\text{CIC}} \tau_1 : \Omega \qquad \Delta;\Gamma, x{:}\tau_1 \vdash \bar{e} : \tau_2}{\Delta;\Gamma \vdash \lambda x \Rightarrow \bar{e} : \mathsf{arw}\ \tau_1\ \tau_2}\ \textit{(I-arw)} \qquad \frac{\Delta;\Gamma \vdash \bar{e}_1 : \mathsf{arw}\ \tau_1\ \tau_2 \qquad \Delta;\Gamma \vdash \bar{e}_2 : \tau_1}{\Delta;\Gamma \vdash \bar{e}_1\ \bar{e}_2 : \tau_2}\ \textit{(E-arw)}$$

$$\frac{\Delta \vdash^{\text{CIC}} \kappa : \mathsf{Kind} \qquad \Delta, t{:}\kappa;\Gamma \vdash \bar{f} : \tau}{\Delta;\Gamma \vdash \bar{f} : \forall t{:}\kappa.\tau}\ \textit{(I-}\forall\textit{)} \qquad \frac{\Delta;\Gamma \vdash \bar{e}_2 : \exists t{:}\kappa.\tau_1 \qquad \Delta;\Gamma \vdash \bar{e}_1 : \forall t{:}\kappa.\mathsf{arw}\ \tau_1\ \tau_2 \qquad \Delta \vdash^{\text{CIC}} \tau_2 : \Omega}{\Delta;\Gamma \vdash \bar{e}_1\ \bar{e}_2 : \tau_2}\ \textit{(E-}\exists\textit{)}$$

Figure 8. Typing rules of λ_{CU}.

Another unusual aspect is that we write part of the proof on paper and part in CiC. Basically, all properties of the coercion calculus need to be proved in CiC (because it is very difficult to take into account every possible interaction with the rest of the CiC system when doing it on paper), whereas the rest of the proof that deals with the term language is written on paper because this part has simply not been formalized in CiC.

LEMMA 6.2 (Substitution).
If $\Delta;\Gamma \vdash \bar{e}_1 : \tau_1$ *and* $\Delta;\Gamma, x{:}\tau_1 \vdash \bar{e}_2 : \tau_2$ *then*
$\Delta;\Gamma \vdash \bar{e}_2[{}^{\bar{e}_1}/_x] : \tau_2$.
If $\Delta \vdash^{\text{CIC}} \varphi : \kappa$ *and* $\Delta, t{:}\kappa;\Gamma \vdash \bar{e} : \tau$ *then*
$\Delta[{}^{\varphi}/_t];\Gamma[{}^{\varphi}/_t] \vdash \bar{e} : \tau[{}^{\varphi}/_t]$.
The proof is straightforward, by induction over the typing derivation.

LEMMA 6.3 (Canonical forms).
If $\bullet;\bullet \vdash \bar{v} : \tau$ *and* $\bullet \vdash^{\text{CIC}} P : \tau \lhd (\mathsf{snat}\ n)$ *then* $\bar{v} \equiv n$.
If $\bullet;\bullet \vdash \bar{v} : \tau$ *and* $\bullet \vdash^{\text{CIC}} P : \tau \lhd (\mathsf{tup}\ n\ \varphi)$ *then*
$\bar{v} \equiv (\bar{v}_0, ..., \bar{v}_{n-1})$.
If $\bullet;\bullet \vdash \bar{v} : \tau$ *and* $\bullet \vdash^{\text{CIC}} P : \tau \lhd (\mathsf{arw}\ \tau_1\ \tau_2)$ *then* $\bar{v} \equiv \lambda x \Rightarrow \bar{e}$.
PROOF SKETCH: E.g. For snat, the proof is by induction on the typing derivation. Since the expression is a value and Γ is empty, the induction will only need to consider the rules *I-snat*, *I-tup*, *I-arw*, *I*-$\forall$, and *Cast*:

- For the first three introduction rules, the conclusion is obtained by proving in CIC that:
$$(\mathsf{snat}\ n1) \lhd (\mathsf{snat}\ n2) \rightarrow N1 = N2$$
$$(\mathsf{tup}\ n\ \varphi) \lhd (\mathsf{snat}\ n) \rightarrow False$$
$$(\mathsf{arw}\ \tau_1\ \tau_2) \lhd (\mathsf{snat}\ n) \rightarrow False$$
- In the case of *I*-$\forall$, we prove in CIC that
$$(\forall t{:}\kappa.\tau) \lhd (\mathsf{snat}\ n) \rightarrow \Sigma w{:}\kappa.\, \tau[{}^{w}/_t] \lhd (\mathsf{snat}\ n)$$
After which we can use the witness w to invoke the induction hypothesis.
- In the case of *Cast*, we can simply compose P with the other coercion into a new coercion and then use the induction hypothesis.

□

LEMMA 6.4 (Type Preservation).
If $\bullet;\bullet \vdash \bar{e} : \tau$ *and* $\bar{e} \rightsquigarrow \bar{e}'$, *then* $\bullet;\bullet \vdash \bar{e}' : \tau$.
PROOF SKETCH: The proof is by case analysis on the reduction step and for each possible step, by induction on the typing derivation. The only unusual part is that the root of the typing derivation may be a string of casts, so we have to go past those until we reach a "real" typing rule that corresponds to the reduction step. The induction steps for the congruence rules are uneventful.

Case $\pi\ i\ (\bar{e}_0, ..., \bar{e}_{n-1}) \Longrightarrow \bar{e}_i$:

Inversion of the typing derivation has two possible cases: *Cast*, which is trivially true by the induction hypothesis, and *E-tup* which gives us that $\bullet;\bullet \vdash i : \mathsf{snat}\ \varphi_1$ and $\bullet;\bullet \vdash (\bar{e}_0, ..., \bar{e}_{n-1}) : \mathsf{tup}\ \varphi_2\ \varphi$ from which we need to show $\bullet;\bullet \vdash \bar{e}_i : \varphi\ \varphi_1$. The canonical forms lemma gives us that $i = \varphi_1$ and $n = \varphi_2$.

By induction on the typing derivation of the tuple, where we only need to consider the *Cast* where we compose all the coercions into a single one, and *I-tup* cases where we end up with: $\bullet;\bullet \vdash (\bar{e}_0, ..., \bar{e}_{n-1}) : \mathsf{tup}\ n'\ \varphi'$ and $\bullet;\bullet \vdash \bar{e}_i : \varphi'\ i$ and $(\mathsf{tup}\ n'\ \varphi') \lhd (\mathsf{tup}\ n\ \varphi)$. Within CIC we can prove:

$$(\mathsf{tup}\ n'\ \varphi') \lhd (\mathsf{tup}\ n\ \varphi) \rightarrow n' = n \wedge \Pi i{:}\mathsf{nat}.\, (\varphi'\ i) \lhd (\varphi\ i)$$

So we can simply use the i^{th} coercion to finally show that $\bullet;\bullet \vdash \bar{e}_i : \varphi\ i$.

Case $(\lambda x \Rightarrow \bar{e}_1)\ \bar{e}_2 \Longrightarrow \bar{e}_1[{}^{\bar{e}_2}/_x]$:

Again, inversion of the typing derivation has two possible cases: *Cast*, which is trivially true by the induction hypothesis, and *E-arw* which gives us $\bullet;\bullet \vdash \lambda x \Rightarrow \bar{e}_1 : \mathsf{arw}\ \tau_1\ \tau_2$ and $\bullet;\bullet \vdash \bar{e}_2 : \tau_1$. Sadly, the first judgment might not come directly from *I-arw*, but instead through a sequence of *Cast* and then *I*-$\forall$ rules. We compose all the casts into a single coercion. We also prove in CIC that:

$$(\forall t{:}\kappa.\tau) \lhd (\mathsf{arw}\ \tau_1\ \tau_2) \rightarrow \Sigma w{:}\kappa.\, \tau[{}^{w}/_t] \lhd (\mathsf{arw}\ \tau_1\ \tau_2)$$

So when we reach a *I*-$\forall$, we can apply the substitution lemma on the antecedent, using the witness w. This repeats until we finally get to an *I-arw* at which point we have:

$$\bullet;\bullet \vdash \lambda x \Rightarrow \bar{e}_1 : \mathsf{arw}\ \tau_1'\ \tau_2'$$
and
$$\bullet \vdash^{\text{CIC}} P : (\mathsf{arw}\ \tau_1'\ \tau_2') \lhd (\mathsf{arw}\ \tau_1\ \tau_2)$$

We can also prove in CIC that

$$(\mathsf{arw}\ \tau_1'\ \tau_2') \lhd (\mathsf{arw}\ \tau_1\ \tau_2) \rightarrow \tau_1 \lhd \tau_1' \wedge \tau_2' \lhd \tau_2$$

so we can use the first coercion to turn $\bullet;\bullet \vdash \bar{e}_2 : \tau_1$ into $\bullet;\bullet \vdash \bar{e}_2 : \tau_1'$, then apply the substitution lemma to get $\bullet;\bullet \vdash \bar{e}_1[{}^{\bar{e}_2}/_x] : \tau_2'$ and finally apply the second coercion to recover $\bullet;\bullet \vdash \bar{e}_1[{}^{\bar{e}_2}/_x] : \tau_2$. □

LEMMA 6.5 (Progress).

If $\bullet; \bullet \vdash \bar{e} : \tau$ *then either* $\bar{e} \rightsquigarrow \bar{e}'$ *or* $\bar{e}$ *is a value.*

The proof is again by induction on the typing derivation, using the canonical forms lemma.

THEOREM 6.6 (Soundness).
If $\Delta; \Gamma \vdash \bar{e} : \tau$*, then either* $\bar{e}$ *is a value, or there exists* $\bar{e}'$ *such that* $\bar{e} \rightsquigarrow \bar{e}'$ *and* $\Delta; \Gamma \vdash \bar{e}' : \tau$.
PROOF: Combine the progress and preservation lemmas. □

A note on the CiC proofs: the proofs were written in Coq, but as it turns out, it seemed basically impossible to write those proofs directly. Instead, the $\lhd$ inductive type is first converted into an alternative representation which is more canonicalized so that there are fewer cases to consider. This moves most of the proof burden to the conversion of ccomp which needs to do a lot of reduction to cancel out and merge the coercions is combines.

7. Related work and discussion

Obviously, the most closely related work is [Crary, 2000], already presented in Sec. 3.

Another related line of research is in [Crary and Weirich, 1999], where they present a coercion called vcase that takes a very different syntactical form, simulating control-flow statements rather than value manipulations.

Of course, all the work surrounding Shao et al.'s type system for certified binaries [Shao et al., 2002, League and Monnier, 2006, Monnier, 2004] was the basis of this work as well.

Systems that provide much more limited forms of coercions are common. The most closely related is probably Concoqtion [Fogarty et al., 2007] which defines a system inspired by λ_H and provides a simple form of coercion.

A different approach was taken in [Hawblitzel et al., 2004] where types are partly brought down to the level of terms, and where the η-redexes and empty loops resulting from free type operations get compiled away via a special optimization phase.

In [Harper and Stone, 2000], Harper and Stone also define the semantics and soundness of a source language (SML) by first translating it into some slightly lower level language which they prove sound.

Future work Our Swiss coercion as presented here covers all of Crary's coercions except for the rec and isorec congruence rules for recursive types (we can easily define intersection types with their coercions on top of λ_{CH}). We have already tried to add crec rules to our coercions which would cover some or all uses of rec and isorec, but we are still struggling with the subsequent soundness proof.

We want to extend the cfold and cunfold rules to allow the use of higher-kinded recursive types. [Collins and Shao, 2002] shows one way to do that.

The let construct that opens existential packages is unnecessary as soon as we can show $\forall t.\mathsf{arw}\ (\varphi\ t)\ \tau \lhd \mathsf{arw}\ (\exists\ \varphi)\ \tau$. Sadly, the current definition of the Swiss coercion is not able to prove it and adding the rule as an additional primitive coercion leads to difficulty in the soundness proof.

References

Gregory D. Collins and Zhong Shao. Intensional analysis of higher-kinded recursive types. Technical Report YALEU/DCS/TR-1240, Yale University, New Haven, CT, 2002.

Karl Crary. Typed compilation of inclusive subtyping. In *International Conference on Functional Programming*, Montréal, Canada, September 2000. ACM Press.

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

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

Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. Extended version published as CMU technical report CMU-CS-97-147.

Chris Hawblitzel, Edward Wei, Heng Huang, Eric Krupski, and Lea Wittie. Low-level linear memory management. In *Informal proceedings of the SPACE Workshop*, Venice, Italy, January 2004.

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

Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of featherweight java. *Transactions on Programming Languages and Systems*, 24(2):112–152, March 2002.

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

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

Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Symposium on Programming Languages Design and Implementation*, pages 81–91, May 2001.

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.

C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*. LNCS 664, Springer-Verlag, 1993.

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.

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.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.