Inductive Types Deconstructed

The Calculus of United Constructions

Stefan Monnier

Université de Montréal - DIRO

Canada

monnier@iro.umontreal.ca

Abstract

Algebraic data types and inductive types like those of the Calculus of Inductive Constructions (CIC) are the bread and butter of statically typed functional programming languages. They conveniently combine in a single package product types, sum types, recursive types, and indexed types. But this also makes them somewhat heavyweight: for example, tuples have to be defined as "degenerate" single constructor inductive types, and extraction of a single field becomes a laborious full case-analysis on the object. We consider this to be unsatisfactory. In this article, we develop an alternative presentation of CIC's inductive types where the various elements are provided separately, such that inductive types are built on top of tuples and sums rather than the other way around. The resulting language is lower-level yet we show it can be translated to to a predicative version of the Calculus of Inductive Constructions in a type-preserving way. An additional benefit is that it can conveniently give a precise type to the default branch of case statements.

CCS Concepts • **Software and its engineering → Functional languages**; *Control structures*; *Syntax*; • **Theory of computation** → *Type theory*.

Keywords Inductive types, compilation, union types, case analysis

ACM Reference Format:

Stefan Monnier. 2019. Inductive Types Deconstructed: The Calculus of United Constructions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3331554.3342607

TyDe '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6815-5/19/08…$15.00
https://doi.org/10.1145/3331554.3342607

1 Introduction

Typer [Monnier 2019] is a functional language based on a dependently typed calculus, in the tradition of Coq [Huet et al. 2000] and Agda [Norell 2007], but focusing on programs more than proofs, like Idris [Brady 2013], F-star [Swamy et al. 2016], Zombie [Casinghino et al. 2014], and many others. Its design follows that of Scheme, in the sense that it intends to provide a minimalist core language on top of which a nice surface language can be built by metaprogramming.

So a focus of Typer's design is on providing a good core language which is the target of the metaprogramming facilities. Some of the design goals of this language are:

1. Usable to write both proofs and programs.
2. An economy of concepts, in other words a simple language with orthogonal features. We want this both for aesthetic and pragmatic reasons.
3. High-level enough to be convenient to build on top of it.
4. Low-level enough so the language itself does not impose unneeded inefficiencies which the compiler then needs to eliminate.
5. A reasonably efficient implementation shouldn't require excessive efforts.

A language like the Calculus of Constructions (CoC) satisfies the first two points above, but falls short on the efficiency side when it comes to representing data structures. The Calculus of Inductive Constructions (CIC) [Paulin-Mohring 1993] solves most of those issues, especially in the form presented by Giménez [1994]. It is nicely minimalist, its inductive definitions providing at once sums types, tuples types, recursive types, etc... For these reasons Typer's core language derives directly from the CIC.

Yet, early experience with it made us feel that it was still a bit too high-level, introducing inefficiencies in some places. For example, it makes it impossible to separate the extraction

of the head and tail of a list from the test that the list is non-empty.

Another annoyance appears in code that wants to manipulate tuples: while defining tuples as single-constructor datatypes is not problematic in terms of type definition or tuple construction, it becomes constraining when extracting data out of those tuples: every access takes the form of a case statement with a single branch that extracts each and every field of the tuple even if only a single one of those fields is needed. So a simple field selection becomes an operation of size proportional to the tuple's size, and like all case statements it requires a potentially complex type annotation.

Of course, most compilers for languages with inductive types or algebraic datatypes typically solve those performance issues by applying corresponding optimizations in the later phases of the compiler where the internal representation is not typed any more, but that goes against our design goal number 4: we want the programmer (and the metaprogramming code) to be able to apply those optimizations by hand without being hindered by the type system.

In the rest of this paper, we hence present the design of the Calculus of United Constructions (CUC) where sums, recursive types, tuples, and indexed types are provided as separate elements, which together can be used to define our beloved inductive types but can also be used separately. The main new primitive is the introduction of a switch construct, which only looks at an object's tag to transfer control to the appropriate branch but doesn't extract any further data, reflecting instead into the type context the knowledge about which tag was found. An important feature of this switch construct is that it fully supports default branches, also reflecting into the type context the fact that some branches were considered but not taken.

While we work within the context of a dependently typed language that can be used as a higher-order logic, the technique we present is applicable to any other language using algebraic datatypes.

This said, when used as a logic, we would want our CUC language to enjoy the same consistency guarantee as the CIC, so we compare the two in Section 5 by giving a translation of CUC terms to CIC.

The contributions of this article are:

- The CUC language, which provides separately sum types, recursive types, tuple types, and equality types, such that they can be combined to provide the functionality traditionally provided by algebraic data types or inductive types, making it better suited as a compiler intermediate language.
- A kind of case-analysis construct where the default branch also gets refined type information witnessing in an efficient way the branches already tested.
- A type-preserving translation of terms from this language to a predicative version of the Calculus of Inductive Constructions.

2 Problem Description

In this section, we briefly present the two problems our design aims to address. A common way to add inductive types to a language is to extend it as follows (mostly taken from Giménez [1994]):

$$
\begin{array}{llll}
\textit{(index)} & i & \in & \mathbb{N} \\
\textit{(term)} & e, b, c, \tau & ::= & \ldots \mid \mathsf{Ind}(x:\tau)\,\langle\vec{c}\rangle \\
 & & & \mid \mathsf{Con}(i, e) \\
 & & & \mid \langle\tau_r\rangle\mathsf{Case}\ e\ \mathsf{of}\ \left\langle\vec{b}\right\rangle \\
 & & & \mid \mathsf{Fix}_i\ x:\tau\ =\ e
\end{array}
$$

Where "$\mathsf{Ind}(x:\tau)\,\langle\vec{c}\rangle$" is the type constructor where "$\vec{c}$" holds the type of each possible constructor, "$\mathsf{Con}(i, e)$" is the value constructor for the i^{th} constructor of the inductive type "e", Case is the eliminator and Fix allows the definition of functions that perform structural recursion on those inductive data types. The "$\mathsf{Ind}(x:\tau)\,\langle\vec{c}\rangle$" constructor is a neat combination of recursive type, sum type, tuple type, and indexed type families.

The main shortcoming of that presentation, for our use, is that Case is a large construct whose elements cannot be taken apart. While it is often perfectly adequate for source code, this lack of fine-granularity can be inconvenient in lower-level code. For example, you cannot first test which constructor was used and only later, if needed, extract some fields from it, because the two operations have to be performed together. Similarly its naive code size and run-time complexity has as lower bound the number of fields of e, which is impractical when selecting a single field from a large tuple, such as a tuple holding all the definitions exported from a module.

For these reasons, we want to introduce tuples separately from inductive types. To do that, we will deconstruct inductive types into their constitutive elements: tuple types, sum types, recursive types, and equality types.

2.1 Native Tuples

Providing algebraic data types and tuples without overlap is not a new problem.

Tagged Sums For example, Standard ML [Milner et al. 1997] solved it years ago by restricting its datatype constructors to carry exactly one element, no more no less. In other words, SML's datatype only provides sum types and recursive types, and tuples are provided separately.

While it is elegant, this solution suffers from an inefficiency we wanted to avoid. For example, with a datatype like:

```
datatype 'a list =
  | cons of 'a * 'a list
  | nil
```

a value like "`cons(1, nil)`" will generally have to be represented as two heap objects: one containing the "`(1, nil)`" tuple and another containing the `cons` tag and a pointer to the tuple. A sufficiently smart compiler may be able to eliminate this indirection, of course, but it can be surprisingly complicated and costly to optimize data representation this way, requiring coercions like those in [Leroy 1992, 1997; Shao 1997], for example if you need to preserve the expected compatibility with a signature which specifies "`cons of 'b`".

First-class Tags Another approach is to let the user manipulate *tags* explicitly as first-class values, making it possible for the user to make their own sum types as tuples whose first field holds a tag and whose subsequent fields have a type that varies according to that tag.

While this approach can be used as well, it comes with both theoretical and practical problems: on the theoretical side, this only works when the universe level of the fields does not depend on the tag; and on the practical side, this requires storing the tag as an extra normal field, whereas in a standard implementation of datatypes, tags are stored typically at no extra cost in a special header word that is needed for memory management purposes anyway.

2.2 Typing the Default Branch

When performing case analysis in Coq and other languages of the family, the default branch does not get any typing refinement. More specifically, let's consider the following example where "`E`" is assumed to be a list:

```
match E with
| nil => EXP1
| _   => EXP2
```

The typing of "`EXP1`" can take advantage of the fact that we know "`E`" was found to be equal to "`nil`", but the typing of "`EXP2`" has no such benefit: it cannot take advantage of the fact that we have found "`E`" to be different from "`nil`".

One might be tempted to solve this problem by changing Coq such that default branches get additional typing information, either providing them with a proof that the match target is different from all the mentioned branches (i.e. a proof that "`not (E = nil)`" in the above example), or providing them with a proof that the match target is equal to one of the remaining possibilities (i.e. a proof that "`∃x,y. E = cons x y`" in the above example).

$$
\begin{array}{lrcl}
\textit{(var)} & x, y, t & \in & \mathcal{V} \\
\textit{(level)} & \ell & \in & \mathbb{N} \\
\textit{(ctxt)} & \Gamma, \Delta & ::= & \bullet \mid \Gamma, x : \tau \\
\textit{(sort)} & s & ::= & \mathsf{Type}_\ell \\
\textit{(term)} & e, \tau & ::= & s \mid x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid (x : \tau_1) \to \tau_2
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S} & = & \{\ \mathsf{Type}_\ell \mid \ell \in \mathbb{N}\ \} \\
\mathcal{A} & = & \{\ (\mathsf{Type}_\ell : \mathsf{Type}_{\ell+1}) \mid \ell \in \mathbb{N}\ \} \\
\mathcal{R} & = & \{\ (\mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\ell_3}) \mid \ell_3 = \ell_1 \sqcup \ell_2\ \}
\end{array}
$$

Figure 1. Our base calculus CCω as a PTS

But this suffers from both theoretical and practical problems: first, while it might seem easy to do it for the source language of Coq, it is not clear how that would work for its core language, where things like "=" and "∃" are themselves encoded as inductive types; second, those additional proofs would tend to grow fairly large.

Since this kind of typing refinement of the default branch is only useful for some fraction of all case analyses, we want a solution that does not incur such excessive extra costs, or more specifically, we want a solution where this refinement is cheap enough that it is practical to provide it everywhere.

2.3 The Base Calculus CCω

Our calculus is built on top of a traditional λ-calculus, and to a first approximation is independent from it, so we will use the same base calculus for both the reference calculus of inductive constructions (CIC) as well as our calculus of united constructions (CUC) which we present in the next few sections in the form of a collection of extensions.

Figure 1 shows our base language CCω as a pure type system (PTS) [Barendregt 1991]. It is a variant of CoC with a tower of universes à la ECC [Luo 1989]. Note that we use the notation "$(x : \tau_1) \to \tau_2$" for the dependent function type, which can of course be shortened to "$\tau_1 \to \tau_2$" when "x" does not occur in "τ_2". We usually use the metavariable "τ" to stand for a term which is supposed to be a type, i.e. whose type is a sort.

Because inductive types have non-trivial interactions with impredicativity, we did not include an impredicative universe at the bottom: all the calculi presented in this paper are fully predicative.

The typing judgment of the base language is denoted "$\Gamma \vdash e : \tau$", and we will annotate it as "$\Gamma \vdash_U e : \tau$" resp. "$\Gamma \vdash_I e : \tau$" when we talk about the typing derivation of CUC resp. CIC. Similarly, while the base language's reduction rule is written "$e \rightsquigarrow e'$", we will write it as "$e \overset{U}{\rightsquigarrow} e'$" or "$e \overset{I}{\rightsquigarrow} e'$" when we talk about the reduction rule for CUC resp. CIC.

$$
\begin{array}{llll}
\textit{(var)} & x, y, t & \in & \mathcal{V} \\
\textit{(level)} & \ell & \in & \mathbb{N} \\
\textit{(label)} & l & \in & \mathcal{L} \\
\textit{(sort)} & s & ::= & \mathsf{Type}_\ell \\
\textit{(ctxt)} & \Gamma, \Delta & ::= & \bullet \mid \Gamma, x : \tau \\
\textit{(term)} & e, \tau, p & ::= & s \mid x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid (x : \tau_1) \to \tau_2 \\
& & \mid & \mathsf{Tuple}_l\ \Delta \mid \mathsf{tuple}_l\ \Delta\ \vec{e} \mid \mathsf{let}\ \vec{x} = e_1\ \mathsf{in}\ e_2 \\
& & \mid & e_1 \equiv e_2 \mid \mathsf{refl}\ e \mid \mathsf{J}\ e_{\equiv}\ e_f\ e \\
& & \mid & \tau_1 \cup \tau_2 \mid \mathsf{cast}\ \tau_1 \subseteq \tau_2\ e \mid \\
& & & (\mathsf{switch}\ e \mid l\ x\ x_{\equiv} \Rightarrow e_l \mid y\ y_{\equiv} \Rightarrow e_d) \\
& & \mid & \mu_i x : \tau . e \mid \mu x : \tau . e \mid \\
& & & \mathsf{fold}\ \tau\ \vec{p}\ e \mid \mathsf{let\ fold}\ \tau\ \vec{p}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

Figure 2. Calculus of united constructions

3 The Calculus of United Constructions

Just like the full CIC, our new calculus is fairly large, so we present it in several steps: the tuples, the equality type, the sums, and finally the recursive definitions. The different parts are not completely independent from each other, which largely dictates the order in which they are presented.

The syntax of the complete language is the following shown in Figure 2. The terms are spread over 5 groups, where the first group reproduces the terms of the base calculus (Sec. 2.3), the second shows the terms of the tuple types (Sec. 3.1), the third shows the terms of the equality types (Sec. 3.2), the fourth shows the terms of the union types (Sec. 3.3), and the last shows the terms of the recursive functions and recursive types (Sec. 3.4).

Clearly, this calculus is much larger than the base calculus. It is also larger than the CIC: as often, there is a tension between keeping the language small and making it efficient. The criteria which drove us to this design favors a larger language as long as the different parts are orthogonal and are themselves simpler or more general.

3.1 Labeled Tuples

At its core, our solution to our design problem is very simple: instead of using "plain" tuples on one side and tagged sums on the other, as is done in SML, we associate the tags (which we call *labels*) with the tuples, so that our sums can be reduced to mere (non-disjoint) union types. The extra cost of adding a label to every tuple is very minor; more specifically in most cases those labels can be stored at no extra cost within the metadata needed for memory management purposes, and even in the worst case it just adds one extra field to those heap objects, which is still much cheaper than the extra heap object and indirection introduced by SML's tagged sums.

$$e.i \equiv \mathsf{let}\ \vec{x} = e\ \mathsf{in}\ x_i \quad \text{(syntactic sugar)}$$

$$\frac{}{\Gamma \vdash_U \mathsf{Tuple}_l\ \bullet : \mathsf{Type}_0}$$

$$\frac{\Gamma \vdash_U \mathsf{Tuple}_l\ \Delta : \mathsf{Type}_{\ell_1} \qquad \Gamma, \Delta \vdash_U \tau : \mathsf{Type}_{\ell_2}}{\Gamma \vdash_U \mathsf{Tuple}_l\ \Delta, x : \tau : \mathsf{Type}_{(\ell_1 \sqcup \ell_2)}}$$

$$\frac{|\vec{e}| = 0}{\Gamma \vdash_U \vec{e} : \bullet} \qquad \frac{\Gamma \vdash_U \vec{e} : \Delta \qquad \Gamma \vdash_U e_i : \tau_i[\vec{e}/\Delta]}{\Gamma \vdash_U \vec{e}, e_i : \Delta, x : \tau_i}$$

$$\frac{\Gamma \vdash_U \vec{e} : \Delta}{\Gamma \vdash_U \mathsf{tuple}_l\ \Delta\ \vec{e} : \mathsf{Tuple}_l\ \Delta}$$

$$\frac{\begin{array}{c}\Gamma \vdash_U e_1 : \mathsf{Tuple}_l\ \Delta \\ \Gamma, \Delta \vdash_U e_2 : \tau \qquad \Delta = x_0 : \tau_0, .., x_n : \tau_n\end{array}}{\Gamma \vdash_U \mathsf{let}\ \vec{x} = e_1\ \mathsf{in}\ e_2 : \tau[e_1.0, ..., e_1.n/x_0, ..., x_n]}$$

Figure 3. Typing rule of labeled tuples

To add tuples to our base language, our language's syntax is extended as follows:

$$
\begin{array}{llll}
\textit{(label)} & l & \in & \mathcal{L} \\
\textit{(term)} & e, \tau & ::= & \ldots \mid \mathsf{Tuple}_l\ \Delta \mid \mathsf{tuple}_l\ \Delta\ \vec{e} \mid \mathsf{let}\ \vec{x} = e_1\ \mathsf{in}\ e_2
\end{array}
$$

"$\mathsf{Tuple}_l\ \Delta$" is the type constructor for tuples with label l where Δ is the list of (possibly dependent) field types; "$\mathsf{tuple}_l\ \Delta\ \vec{e}$" is the introduction form which lets you actually build tuple values; and "$\mathsf{let}\ \vec{x} = e_1\ \mathsf{in}\ e_2$" is the eliminator which extracts the values of the tuple e_1. Figure 3 shows the typing rules for our labeled tuples. In there we also define a projection "$e.i$" as syntactic sugar for "$\mathsf{let}\ \vec{x} = e\ \mathsf{in}\ x_i$", although it could also be provided as a built-in construct. We will discuss the choice of elimination forms in Sec. 6. Reduction rules of the language are extended with the obvious congruence rules as well as the following primitive reduction:

$$\mathsf{let}\ \vec{x} = (\mathsf{tuple}_l\ \_\ \vec{e})\ \mathsf{in}\ e \overset{U}{\rightsquigarrow} e[\vec{e}/\vec{x}]$$

The form of our tuple constructor "$\mathsf{tuple}_l\ \Delta\ \vec{e}$" was chosen to be "saturated" in the sense that all elements of the tuple have to be provided, rather than providing a tuple constructor which only takes the Δ argument and then receives the tuple elements in a curried fashion, as is done for example in CIC's inductive constructors and in Haskell's datatype constructors. This was done for two reasons: most importantly, it makes the construct directly correspond to the actual allocation and initialization of the heap object, so the cost of any extra closures needed for curried use have to be made explicit in the code; second it preserves the property that any value of arrow type has to be of the form "$\lambda x : \tau . e$". This second point turned out to be unimportant: not only we do

$$\frac{\Gamma \vdash_U e_1 : \tau \qquad \Gamma \vdash_U e_2 : \tau \qquad \Gamma \vdash_U \tau : \mathsf{Type}_\ell}{\Gamma \vdash_U e_1 \equiv e_2 : \mathsf{Type}_\ell}$$

$$\frac{\Gamma \vdash_U e : \tau}{\Gamma \vdash_U \mathsf{refl}\ e : e \equiv e}$$

$$\frac{\Gamma \vdash_U e_\equiv : e_1 \equiv e_2 \qquad \Gamma \vdash_U e_f\ e_1 : s \qquad \Gamma \vdash_U e : e_f\ e_1}{\Gamma \vdash_U \mathsf{J}\ e_\equiv\ e_f\ e : e_f\ e_2}$$

Figure 4. Typing rules of the equality types

not make use of this property, but later parts of our system break it anyway.

3.2 Equality

Armed with tuples, we can now do most of what is done with single-constructor non-recursive types, but not all: our tuples do not offer us any way to define the equivalent of those single-constructor inductive types which are *indexed*. The main example of such a type is the equality type. In CoC, the equality type can be defined using the impredicative encoding, with the usual associated restrictions, but our base calculus being predicative we don't even have that option. So we extend our language with an equality type:

$$\textit{(term)} \quad e, \tau \ ::= \ \ldots \mid e_1 \equiv e_2 \mid \mathsf{refl}\ e \mid \mathsf{J}\ e_\equiv\ e_f\ e$$

$\equiv$ is the type constructor for this new equality type; *refl* is the corresponding introduction form, and J its eliminator which encodes the Leibniz equality. Figure 4 shows the corresponding typing rules. The corresponding new primitive reduction rule is the following:

$$\mathsf{J}\ (\mathsf{refl}\ \_)\ \_\ x \overset{U}{\rightsquigarrow} x$$

where the underscores represent subterms which are ignored by the rule.

While the equality type was not the only single-constructor inductive type we could not define, we can now define also all the other single-constructor inductive types by adding appropriate fields holding proofs of the needed type equalities.

For example, in a language with indexed inductive types, we can define the traditional length-indexed list type as follows:

```
type NList (α : Type) : Nat -> Type
  | nil  : NList α 0
  | cons : α -> NList α l -> NList α (S l);
```

In the absence of indexed inductive types this can be easily replaced by the following definition which uses explicit equality witnesses instead of indices:

```
type NList (α : Type) (l : Nat) : Type
```

$$\frac{\Gamma \vdash_U \tau_1 : \mathsf{Type}_{\ell_1} \qquad \Gamma \vdash_U \tau_1 : \mathsf{Type}_{\ell_1}}{\Gamma \vdash_U \tau_1 \cup \tau_2 : \mathsf{Type}_{\ell_1 \sqcup \ell_2}}$$

$$\frac{\Gamma \vdash_U \tau : s \qquad \Gamma \vdash_U e : \tau_e \qquad \tau_e \subseteq \tau}{\Gamma \vdash_U \mathsf{cast}\ \tau_e \subseteq \tau\ e : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash_U e : \tau_e \qquad \tau_e \overset{l}{\Longrightarrow} \tau_l \cup \tau_d \qquad \tau_l \neq \bot \qquad \tau_d \neq \bot \\ \Gamma, x : \tau_l, x_\equiv : (e \equiv \mathsf{cast}\ \tau_l \subseteq \tau_e\ x) \vdash_U e_l : \tau_r \\ \Gamma, y : \tau_d, y_\equiv : (e \equiv \mathsf{cast}\ \tau_d \subseteq \tau_e\ y) \vdash_U e_d : \tau_r\end{array}}{\Gamma \vdash_U \mathsf{switch}\ e \mid l\ x\ x_\equiv \Rightarrow e_l \mid y\ y_\equiv \Rightarrow e_d : \tau_r}$$

Figure 5. Typing rules for union types

```
  | nil  : (l ≡ 0) -> NList α l
  | cons : α -> NList α l'
           -> (l ≡ S l') -> NList α l;
```

The absence of indexed types also has the benefit that case analysis does not need to make special allowances to support *type refinement* of the indices: the explicit equality witnesses can be used to get the same effect. The same approach is used in GHC, of course [Sulzmann et al. 2007].

3.3 Unions

Since our tuples carry labels, we can rely on those to discriminate between alternatives of sum types, which frees us from the need to use disjoint unions and instead we can use the leaner untagged union types. We extend the syntax with a new union type as well as corresponding introduction and elimination constructs:

$$\begin{array}{rl}\textit{(term)} \quad e, \tau \ ::= \ \ldots & \mid \tau_1 \cup \tau_2 \\ & \mid \mathsf{cast}\ \tau_1 \subseteq \tau_2\ e \\ & \mid \mathsf{switch}\ e \\ & \quad \mid l\ x\ x_\equiv \Rightarrow e_l \\ & \quad \mid y\ y_\equiv \Rightarrow e_d\end{array}$$

$\cup$ is the type constructor for unions; The "cast $\tau_1 \subseteq \tau_2\ e$" operation is the corresponding introduction form; it should be read as a form of weakening of "e" from a subtype "τ_1" to a supertype "τ_2", at no run-time cost; while the switch construct "switch $e \mid l\ x\ x_\equiv \Rightarrow e_l \mid y\ y_\equiv \Rightarrow e_d$" is the elimination form, which lets us recover the corresponding information, with a run-time cost comparable to that of a C switch: it looks at the label of tuple "e" and jumps to "e_l" if it's equal to "l" and to "e_d" otherwise, but it does not perform any further extraction of data. In the switch's branches, "$x/x_\equiv$" and "$y/y_\equiv$" are pairs of variables which get bound respectively to the value of "e" strengthened to a more specific type and to a proof that this new variable is indeed just a strengthened "e".

Figure 5 shows the typing rules for unions. These introduce three new rules, one per construct, added to the main typing

$$\text{cast } \_ \subseteq \tau_2\ (\text{cast } \tau_1 \subseteq \_\ e) \overset{U}{\rightsquigarrow} \text{cast } \tau_1 \subseteq \tau_2\ e \qquad \text{cast } \tau \subseteq \tau\ e \overset{U}{\rightsquigarrow} e$$

$$\frac{e = \text{cast } \tau \subseteq \tau_e\ e' \qquad e' = \text{tuple}_{l'}\ \Delta\ \vec{e} \qquad \tau_e \overset{l}{\Longrightarrow} \tau_l \cup \tau_d}{\text{switch } e \mid l\ x\ x_{\equiv} \Rightarrow e_l \mid y\ y_{\equiv} \Rightarrow e_d \overset{U}{\rightsquigarrow} \begin{cases} e_l[\text{refl } e, \text{cast } \tau \subseteq \tau_l\ e'/x_{\equiv}, x] & \text{if } l = l' \\ e_d[\text{refl } e, \text{cast } \tau \subseteq \tau_d\ e'/y_{\equiv}, y] & \text{otherwise} \end{cases}}$$

Figure 6. Reduction rules for the union types

$\tau \overset{l}{\Longrightarrow} \tau_l \cup \tau_d$	Split τ according to l
$\tau_1 \cup' \tau_2$	Like $\cup$ but eliminating $\bot$
$\tau_1 \subseteq \tau_2$	τ_1 is a subtype of τ_2

$$\frac{}{\text{Tuple}_l\ \Delta \overset{l}{\Longrightarrow} \text{Tuple}_l\ \Delta \cup \bot}$$

$$\frac{\tau_1 \overset{l}{\Longrightarrow} \tau_{l1} \cup \tau_{d1} \qquad \tau_2 \overset{l}{\Longrightarrow} \tau_{l2} \cup \tau_{d2}}{\tau_1 \cup \tau_2 \overset{l}{\Longrightarrow} (\tau_{l1} \cup' \tau_{l2}) \cup (\tau_{d1} \cup' \tau_{d2})}$$

$$\frac{l' \neq l}{\text{Tuple}_{l'}\ \Delta \overset{l}{\Longrightarrow} \bot \cup \text{Tuple}_{l'}\ \Delta}$$

$$\bot \cup' \tau = \tau \qquad \tau \cup' \bot = \tau \qquad \tau_1 \cup' \tau_2 = \tau_1 \cup \tau_2$$

$$\frac{}{\tau \subseteq \tau} \qquad \frac{\tau_1 \subseteq \tau_2}{\tau_1 \subseteq \tau_2 \cup \tau_3} \qquad \frac{\tau_1 \subseteq \tau_3}{\tau_1 \subseteq \tau_2 \cup \tau_3}$$

$$\frac{\tau_1 \subseteq \tau_3 \qquad \tau_2 \subseteq \tau_3}{\tau_1 \cup \tau_2 \subseteq \tau_3}$$

Figure 7. Auxiliary rules for union types

judgment and they rely on three auxiliary judgments presented in Figure 7: the subtype relation "$\tau_1 \subseteq \tau_2$" used for cast; the "smart constructor" "$\tau_1 \cup' \tau_2$" which is like $\cup$ except it tries to eliminate the "$\bot$" elements which might have been introduced; and finally "$\tau \overset{l}{\Longrightarrow} \tau_l \cup \tau_d$" which plays two roles. First, it is used to ensure that switch is only applied to (weakened) tuples, which is indispensable at run-time so that we can safely go fetch the object's label even though its type is a union type rather than a tuple type. Second, it is used to find the refined type of "e" in each branch, splitting "τ" into the part "τ_l" that matches the label "l" and the part "τ_d" which does not.

As presented, our switch statement has the unusual property that it tests a single label before falling through to a default branch, but since the default branch properly preserves the information that this label failed to match, it can be trivially chained in order to select between several possible labels. Also it is straightforward to extend the language to allow the presence of an arbitrary number of branches before the default branch, of course.

Another unusual property of this switch statement, compared to the case analysis rule of traditional inductive types is that the return type of all branches is the same: the type refinement needed for dependent elimination is replaced by the explicit equality proof bound to $x_{\equiv}$ or $y_{\equiv}$ witnessing which branch was chosen.

Reduction rules of the language are extended with the obvious congruence rules as well as the primitive reductions shown in Figure 6. These reduction rules are fairly complex to our taste, especially compared to something like the β rule, but they will be simplified in the erasure semantics presented in Section 4.

3.4 Recursion

The final missing component of inductive types is that which gives it its name: the ability to define recursive types and to perform induction on values of those types. Both of those correspond to forms of recursive definitions, one of them for types and the other for functions. We could handle both cases within the same fixpoint construct, but since they require different termination checking rules, we have kept the two syntactically separate. Concretely, the syntax is extended as follows:

$$\begin{array}{llll} \textit{(index)} & i & \in & \mathbb{N} \\ \textit{(term)} & e, \tau, p & ::= & \ldots \mid \mu_i x : \tau . e \mid \mu x : \tau . e \\ & & & \mid \text{fold } \tau\ \vec{p}\ e \mid \text{let fold } \tau\ \vec{p}\ x = e_1 \text{ in } e_2 \end{array}$$

$\mu x : \tau . e$ is the fixpoint construct that can be used to define recursive types, as long as they obey the customary strict positivity constraint; $\mu_i x : \tau . e$ is the fixpoint construct that can be used to define recursive functions when they abide by a syntactic restriction that ensures that the i^{th} argument becomes smaller at each recursive call. The $\mu x : \tau . e$ type

$$\text{unfold } \tau\ \vec{p}\ e \ \equiv\ \text{let fold } \tau\ \vec{p}\ x = e \text{ in } x \quad \text{(syntactic sugar)}$$

$$\frac{\Gamma \vdash_U \tau : s \qquad \Gamma, x:\tau \vdash_U e : \tau \qquad e = \lambda\overrightarrow{z:\tau_z}.e' \qquad x \notin \text{fv}(\vec{\tau}) \qquad x \vdash_U e' \ \text{pos}}{\Gamma \vdash_U \mu x:\tau.e : \tau}$$

$$\frac{\Gamma \vdash_U \tau : s \qquad \Gamma, x:\tau \vdash_U e : \tau \qquad e = \lambda\overrightarrow{y:\_}.e_b \qquad i < |y| \qquad x; i; y_i; \emptyset \vdash_U e_b \ \text{term}}{\Gamma \vdash_U \mu_i x:\tau.e : \tau}$$

$$\frac{\Gamma, \overrightarrow{z:\tau_z}, y:(e_\tau[\tau/x])\ \vec{z} \vdash_U e_2 : \tau_2 \qquad \vec{z} \cap \text{fv}(e_2) = \emptyset \qquad \Gamma \vdash_U \tau : \overrightarrow{(z:\tau_z)} \to s \qquad \tau = \mu x:\_.e_\tau \qquad \Gamma \vdash_U e_1 : \tau\ \vec{p}}{\Gamma \vdash_U \text{let fold } \tau\ \vec{p}\ y = e_1 \text{ in } e_2 : \tau_2[\vec{p}, \text{unfold } \tau\ \vec{p}\ e_1/\vec{z}, y]}$$

$$\frac{\Gamma \vdash_U e : (e_\tau[\tau/x])\ \vec{p} \qquad \Gamma \vdash_U \tau : \_ \qquad \tau = \mu x:\_.e_\tau}{\Gamma \vdash_U \text{fold } \tau\ \vec{p}\ e : \tau\ \vec{p}}$$

Figure 8. Recursive definitions

constructor has corresponding introduction and elimination forms to fold and unfold the recursion.

As usual, additionally to the congruence rules, the reduction rules are extended with the following primitive reduction:

$$\text{let fold } \_\ \_\ x = (\text{fold } \_\ \_\ e_1) \text{ in } e_2 \ \overset{U}{\rightsquigarrow}\ e_2[e_1/x]$$

$$\frac{e_i = \text{tuple}_l\ \_\ \_ \ \vee\ e_i = \text{cast } \_\subseteq\_\ (\text{tuple}_l\ \_\ \_)}{(\mu_i x:\tau.e)\ \vec{e} \ \overset{U}{\rightsquigarrow}\ (e[\mu_i x:\tau.e/x])\ \vec{e}}$$

The first is the usual β-like application of an elimination operation on the corresponding introduction, but the second is less usual: $\mu_i x:\tau.e$ is a value constructor with no matching type constructor nor elimination construct. Its elimination rule is designed to carefully unfold the function often enough not to get in the way, but not too often to cause infinite unfoldings. This rule is directly adapted from the corresponding one presented by Giménez [1994].

The typing rules are given in Figure 8. As was the case for tuples, these include the definition of syntactic sugar, here of the form unfold $\tau\ \vec{p}\ e$. As was the case for union types, the figure shows first the four new rules added to the main typing judgment, one for each new construct, and rely on some auxiliary judgments, shown in Figure 9: first the $x \vdash_U \tau$ pos judgment, which enforces the strict positivity rule on $\mu x:\tau.e$, and then the $x_f; i; x_i; v \vdash_U e$ term judgment which enforces termination of $\mu_i x_f:\tau.e$.

The positivity and termination check are adapted from that of Giménez [1994]. Like his, our positivity rule for our (dependent) tuples enforces that we cannot have a dependence on a recursive argument: if field x_i refers to the recursive

$x \vdash_U e$ pos | e is positive in x

$$\frac{x \notin \text{fv}(e)}{x \vdash_U e \ \text{pos}} \qquad \frac{x \notin \text{fv}(\vec{e})}{x \vdash_U x\ \vec{e} \ \text{pos}}$$

$$\frac{x \vdash_U e \ \text{pos} \qquad x \notin \text{fv}(\tau)}{x \vdash_U (y:\tau) \to e \ \text{pos}}$$

$$\frac{\Delta = x_0:\tau_0, .., x_n:\tau_n \qquad \forall i. \quad x \notin \text{fv}(\tau_i) \ \vee\ (x \vdash_U \tau_i \ \text{pos} \ \wedge\ \forall j > i.\ x_i \notin \text{fv}(\tau_j))}{x \vdash_U \text{Tuple}_l\ \Delta \ \text{pos}}$$

$$\frac{x \vdash_U \tau_1 \ \text{pos} \qquad x \vdash_U \tau_2 \ \text{pos}}{x \vdash_U \tau_1 \cup \tau_2 \ \text{pos}} \qquad \frac{x \vdash_U e \ \text{pos} \qquad x \notin \text{fv}(\tau)}{x \vdash_U \mu y:\tau.e \ \text{pos}}$$

$v \vdash_U e$ small $x_f; i; x_i; v \vdash_U e$ term	e is small assuming v are small all i^{th} args of x_f smaller than x_i in e, given that v are smaller

$$\frac{x \in v}{v \vdash_U x \ \text{small}} \qquad \frac{v \vdash_U e \ \text{small}}{v \vdash_U (e\ \_) \ \text{small}}$$

$$\frac{x_f \notin \text{fv}(e)}{x_f; i; x_i; v \vdash_U e \ \text{term}}$$

$$\frac{x_f; i; x_i; v \vdash_U \vec{e} \ \text{term} \qquad i < |e| \qquad v \vdash_U e_i \ \text{small}}{x_f; i; x_i; v \vdash_U x_f\ \vec{e} \ \text{term}}$$

$$\frac{x_f; i; x_i; v \vdash_U e_1 \ \text{term} \qquad v \cup \{x_i\} \vdash_U e_1 \ \text{small} \qquad x_f; i; x_i; \vec{y} \cup v \vdash_U e_2 \ \text{term}}{x_f; i; x_i; v \vdash_U \text{let } \vec{y} = e_1 \text{ in } e_2 \ \text{term}}$$

$$\frac{x_f; i; x_i; v \vdash_U e_1 \ \text{term} \qquad x_f; i; x_i; v \vdash_U \tau\ \vec{p} \ \text{term} \qquad v \cup \{x_i\} \vdash_U e_1 \ \text{small} \qquad x_f; i; x_i; \{y\} \cup v \vdash_U e_2 \ \text{term}}{x_f; i; x_i; v \vdash_U \text{let fold } \tau\ \vec{p}\ y = e_1 \text{ in } e_2 \ \text{term}}$$

$$\frac{x_f; i; x_i; v \vdash_U e \ \text{term} \qquad x_f; i; x_i; \{x\} \cup v \vdash_U e_l \ \text{term} \qquad v \cup \{x_i\} \vdash_U e \ \text{small} \qquad x_f; i; x_i; \{y\} \cup v \vdash_U e_d \ \text{term}}{x_f; i; x_i; v \vdash_U \text{switch } e \mid l\ x\ x_{\equiv} \Rightarrow e_l \mid y\ y_{\equiv} \Rightarrow e_d \ \text{term}}$$

Figure 9. Termination conditions

argument x, then subsequent fields cannot depend on x_i (i.e. the type of subsequent fields cannot refer to x_i). Contrary to his, we allow nested recursive definitions (in the last rule); and notice that it does not check that those are themselves positive, because this verification will have been done already while type checking them.

We do not show the usual congruence in the rules for the termination check. This check considers switch operations, field extractions from tuples, unfoldings, as well as function calls, as those operations which return something smaller. ν keeps track of *variables* being smaller than the original argument, while $\nu \vdash_U e$ small tests the same for expressions. This same problem applies to the case of function calls, of course, but it is solved differently for them. This irregularity is solely here to better match the rules used in CIC so as to make it easier to see the correspondence between the two.

4 Erasure

The intention of our calculus is for "cast" to have no run-time cost. In this section, we show that it is indeed the case, by defining an erasure function and showing that the evaluation and the erasure commute. The erasure function $(\cdot)*$ is defined recursively on the syntax of terms in a straightforward manner:

$$\begin{array}{ll} x* & \mapsto x \\ (e_1\ e_2)* & \mapsto e_1*\ e_2* \\ ((x:\tau_1) \to \tau_2)* & \mapsto (x:\tau_1*) \to \tau_2* \\ (\lambda x:\tau.e)* & \mapsto \lambda x:\tau*.e* \\ \quad \cdots & \mapsto \cdots \\ (\text{cast } \tau_1 \subseteq \tau_2\ e)* & \mapsto e* \end{array}$$

The $\cdots$ stand for all the remaining constructs where the function simply recurses in the obvious way on all subterms. We do not introduce a new syntax for erased terms because they simply use a subset of the syntax of the non-erased terms. On the other hand, we do need to introduce a new reduction judgment $e_1 \overset{e}{\rightsquigarrow} e_2$. Other than the usual congruence rules, the reduction rules of the erased calculus are the following:

$$(\lambda x:\_.e_1)\ e_2 \overset{e}{\rightsquigarrow} e_1[e_2/x] \qquad \text{J (refl \_) \_ } x \overset{e}{\rightsquigarrow} x$$

$$\text{let } \vec{x} = (\text{tuple}_l\ \_\ \vec{e}) \text{ in } e \overset{e}{\rightsquigarrow} e[\vec{e}/\vec{x}]$$

$$\text{let fold \_ \_ } x = (\text{fold \_ \_ } e_1) \text{ in } e_2 \overset{e}{\rightsquigarrow} e_2[e_1/x]$$

$$\frac{e = \text{tuple}_{l'}\ \_\ \_}{\text{switch } e \mid l\ x\ x_{\equiv} \Rightarrow e_l \mid y\ y_{\equiv} \Rightarrow e_d \overset{e}{\rightsquigarrow} \begin{cases} e_l[\text{refl } e, e/x_{\equiv}, x] & \text{if } l = l' \\ e_d[\text{refl } e, e/y_{\equiv}, y] & \text{otherwise} \end{cases}}$$

$$\frac{e = \mu_i x:\_.e' \qquad e_i = \text{tuple}_l\ \_\ \_}{e\ \vec{e} \overset{e}{\rightsquigarrow} (e'[e/x])\ \vec{e}}$$

Note that the first few rules are taken unmodified from $\overset{U}{\rightsquigarrow}$, and only the rules for switch and for the unfolding of recursive functions are affected by the erasure of cast.

We can show that this erasure calculus is consistent with the original calculus, and hence that we can safely implement cast as a no-op:

Lemma 4.1 (Erasure). *Assuming $\Gamma \vdash_U e_1 : \tau$, we have:*

- *If $e_1 \overset{U}{\rightsquigarrow} e_2$ then either $e_1* = e_2*$ or $e_1* \overset{e}{\rightsquigarrow} e_2*$.*
- *If $e_1* \overset{e}{\rightsquigarrow} e_2$ then there exists an e_3 such that $e_3* = e_2$ and $e_1 \overset{U}{\rightsquigarrow}{}^* e_3$.*

Proof. By induction on the derivation of $e_1 \overset{U}{\rightsquigarrow} e_2$ resp. $e_1* \overset{e}{\rightsquigarrow} e_2$. □

5 Type Soundness

Now that we have defined a calculus which provides us with the intended run-time cost, we show that this calculus is sound. Instead of showing directly, we showing it to be sound with respect to a more classical presentation of inductive types. We will first present a predicative CIC as an extension of the base calculus shown in Sec. 2 with inductive types in the style of Giménez [1994], and then show that any expression of our CUC can be compiled to this CIC.

5.1 Inductive Types

We add inductive types following the style of Giménez [1994] which separates induction into case analysis and recursive definitions, combined with a syntactic check that the recursive calls correspond to a structural induction. The syntax of the base language is extended as follows:

$$\begin{array}{llll} \textit{(index)} & i & \in & \mathbb{N} \\ \textit{(term)} & e, b, c, \tau & ::= & \ldots \mid \text{Ind}(x:\tau)\ \langle \vec{c} \rangle \\ & & & \mid \text{Con}(i, e) \\ & & & \mid \langle \tau_r \rangle \text{Case } e \text{ of } \left\langle \vec{b} \right\rangle \\ & & & \mid \text{Fix}_i\ x : \tau\ =\ e \end{array}$$

$\text{Ind}(x:\tau)\ \langle \vec{c} \rangle$ is an inductive type of kind τ with $|\vec{c}|$ constructors where c_i is the type of the i^{th} constructor; $\text{Con}(i, e)$ is the i^{th} constructor of the inductive type e; $\langle \tau_r \rangle \text{Case } e \text{ of } \left\langle \vec{b} \right\rangle$ performs case analysis on an object e of inductive type; for an object built with the i^{th} constructor, branch b_i will be called, passing to it the arguments that were passed to the constructor; and finally $\text{Fix}_i\ x : \tau\ =\ e$ defines a recursive function which performs a structural induction on its i^{th} argument.

Figure 10 shows the new typing rules and reduction rules introduced for those inductive types, as well as new auxiliary judgments to enforce that inductive types are strictly positive and that recursive definitions are terminating. Beside the slightly different syntax (and the congruence rules we do not show), our system differs from that of Giménez [1994] in the following aspects:

$\Gamma \vdash_I e : \tau$	e has type τ in Γ
$e \overset{I}{\rightsquigarrow} e'$	e reduces to e'

$$\frac{e = \mathsf{Ind}(x:\tau)\,\langle \vec{c} \rangle \qquad \Gamma \vdash_I e : \tau}{\Gamma \vdash_I \mathsf{Con}(i, e) : c_i[x/e]}$$

$$\frac{\Gamma \vdash_I \tau : \mathsf{Type}_{\ell+1} \qquad \tau = \overrightarrow{(\_:\_)} \to \mathsf{Type}_\ell \qquad \forall i. \qquad \Gamma, x:\tau \vdash_I c_i : \mathsf{Type}_\ell \qquad x; x \vdash_I c_i \ \ \mathsf{con}}{\Gamma \vdash_I \mathsf{Ind}(x:\tau)\,\langle \vec{c} \rangle : \tau}$$

$$\frac{\begin{array}{l}\Gamma \vdash_I e : \tau_I\ \vec{p} \qquad \tau_I = \mathsf{Ind}(x:\overrightarrow{(z:\tau_z)} \to s)\,\langle \vec{c} \rangle \qquad \Gamma \vdash_I \tau_r : \overrightarrow{(z:\tau_z)} \to \tau_I\ \vec{z} \to s \\ \forall i. \qquad c_i = \overrightarrow{(y:\tau_y)} \to x\ \vec{p}' \qquad \Gamma \vdash_I b_i : \overrightarrow{(y:\tau_y[\tau_I/x])} \to \tau_r\ \vec{p}'\ (\mathsf{Con}(i, \tau_I)\ \vec{y})\end{array}}{\Gamma \vdash_I \langle \tau_r \rangle \mathsf{Case}\ e\ \mathsf{of}\ \left\langle \vec{b} \right\rangle : \tau_r\ \vec{p}\ e}$$

$$\frac{\Gamma \vdash_I \tau : s \qquad \Gamma, x_f:\tau \vdash_I e : \tau \qquad e = \lambda\overrightarrow{y:\_}.\lambda x_i:\_.e_b \qquad i = |y| \qquad x_f; i; x_i; \emptyset \vdash_U e_b \ \ \mathsf{term}}{\Gamma \vdash_I \mathsf{Fix}_i\ x_f : \tau\ =\ e : \tau}$$

$$\langle \tau_r \rangle \mathsf{Case}\ (\mathsf{Con}(i, e)\ \vec{e})\ \mathsf{of}\ \left\langle \vec{b} \right\rangle \overset{I}{\rightsquigarrow} b_i\ \vec{e} \qquad \frac{i < |\vec{e}| \qquad e_i = \mathsf{Con}(\_, \_) \qquad e = \mathsf{Fix}_i\ x : \tau\ =\ e_f}{e\ \vec{e} \overset{I}{\rightsquigarrow} (e_f[e/x])\ \vec{e}}$$

$x \vdash_I e \ \ \mathsf{pos}$	e is positive in x
$x; y \vdash_I e \ \ \mathsf{con}$	e is the type of a constructor of y, positive in x

$$\frac{x \notin \mathsf{fv}(\vec{e})}{x \vdash_I x\ \vec{e} \ \ \mathsf{pos}} \qquad \frac{x \vdash_I e \ \ \mathsf{pos} \qquad x \notin \mathsf{fv}(\tau)}{x \vdash_I (y:\tau) \to e \ \ \mathsf{pos}} \qquad \frac{x \notin \mathsf{fv}(\vec{e}) \qquad x \notin \mathsf{fv}(\tau) \qquad \forall i. \qquad x; y \vdash_I c_i \ \ \mathsf{con}}{x \vdash_I (\mathsf{Ind}(y:\tau)\,\langle \vec{c} \rangle)\ \vec{e} \ \ \mathsf{pos}}$$

$$\frac{x \notin \mathsf{fv}(\vec{e})}{x; y \vdash_I y\ \vec{e} \ \ \mathsf{con}} \qquad \frac{x; y \vdash_I e \ \ \mathsf{con} \qquad x \notin \mathsf{fv}(\tau)}{x; y \vdash_I (z:\tau) \to e \ \ \mathsf{con}} \qquad \frac{x; y \vdash_I e \ \ \mathsf{con} \qquad x \vdash_I \tau \ \ \mathsf{pos}}{x; y \vdash_I \tau \to e \ \ \mathsf{con}}$$

$v \vdash_I e \ \ \mathsf{small}$	e is small assuming v are small
$x_f; i; x_i; v \vdash_I e \ \ \mathsf{term}$	i^{th} arg of x_f always smaller than x_i in e, given that v are smaller

$$\frac{x \in v}{v \vdash_I x \ \ \mathsf{small}} \qquad \frac{v \vdash_I e \ \ \mathsf{small}}{v \vdash_I (e\ \_) \ \ \mathsf{small}} \qquad \frac{x_f \notin \mathsf{fv}(e)}{x_f; i; x_i; v \vdash_I e \ \ \mathsf{term}} \qquad \frac{x_f; i; x_i; v \vdash_I \vec{e} \ \ \mathsf{term} \qquad i < |e| \qquad v \vdash_I e_i \ \ \mathsf{small}}{x_f; i; x_i; v \vdash_I x_f\ \vec{e} \ \ \mathsf{term}}$$

$$\frac{\begin{array}{l}x_f; i; x_i; v \vdash_I e \ \ \mathsf{term} \qquad x_f; i; x_i; v \vdash_I \tau_r \ \ \mathsf{term} \qquad x_f; i; x_i; v \vdash_I \tau_e \ \ \mathsf{term} \\ v \cup \{x_i\} \vdash_I e \ \ \mathsf{small} \qquad \forall i. \qquad b_i = \lambda\overrightarrow{y:\_}.e_i \qquad x_f; i; x_i; v \cup \vec{y} \vdash_I b_i \ \ \mathsf{term}\end{array}}{x_f; i; x_i; v \vdash_I \langle \tau_r \rangle \mathsf{Case}\ e\ \mathsf{of}\ \left\langle \vec{b} \right\rangle \ \ \mathsf{term}}$$

Figure 10. Inductive types

- Our rules are extended to a tower of universes and the typing rule of Ind enforces predicativity;
- Giménez does not include the Ind rule of $x \vdash_I e \ \ \mathsf{pos}$; which allows us to define for example an inductive type t where one of the fields has type *List t*. Most proof assistants allow such a relaxation of the positivity requirement, and we use this flexibility in our encoding.
- The termination check on Case is simpler in that it considers all fields of an object to be smaller than the object analyzed, whereas Giménez limits this to the fields which are in a *recursive position*. Giménez needs this additional restriction because his Set universe is impredicative, so

$$
\begin{array}{ll}
\mathsf{Unit} & = \mathsf{Ind}(x:\mathsf{Type}_0)\,\langle x\rangle \\
\mathsf{unit} & = \mathsf{Con}(0, \mathsf{Unit}) \\
\Sigma x:\tau_1.\tau_2 & = \mathsf{Ind}(y:\mathsf{Type}_?)\,\langle (x:\tau_1) \to \tau_2 \to y\rangle \\
\langle x = e_1, e_2 : \tau_2\rangle & = \mathsf{Con}(0, \Sigma x:\tau_1.\tau_2)\ e_1\ e_2 \\
\mathsf{eq}\ e_1 & = \mathsf{Ind}(x:? \to x)\,\langle x\ e_1\rangle \\
\mathsf{refl}\ e_1 & = \mathsf{Con}(0, \mathsf{eq}\ e_1) \\
\mathsf{J}\ e_{\equiv}\ e_f\ e & = \langle \lambda x:?.\lambda\_:?.e_f\ x\rangle \mathsf{Case}\ e_{\equiv}\ \mathsf{of}\ \langle e\rangle \\
\mathsf{Either}\ \tau_1\ \tau_2 & = \mathsf{Ind}(x:\mathsf{Type}_?)\,\langle \tau_1 \to x, \tau_1 \to x\rangle
\end{array}
$$

$$
\begin{array}{ll}
[\![\tau\ \subseteq\ \tau]\!]_I & = \lambda x:[\![\tau]\!]_I.x \\
[\![\tau_1\ \cup\ \tau_2\ \subseteq\ \tau_3]\!]_I & = \lambda x:\mathsf{Either}\ [\![\tau_1]\!]_I\ [\![\tau_2]\!]_I\,. \\
 & \quad \langle \_:?.[\![\tau_3]\!]_I\rangle \mathsf{Case}\ x\ \mathsf{of} \\
 & \quad\quad \left\langle \begin{array}{l} [\![\tau_1\ \subseteq\ \tau_3]\!]_I \\ [\![\tau_2\ \subseteq\ \tau_3]\!]_I \end{array} \right\rangle \\
[\![\tau_1\ \subseteq\ \tau_2\ \cup\ \tau_3]\!]_I & = \lambda x:\tau_1.\mathsf{Con}(0, \mathsf{Either}\ [\![\tau_2]\!]_I\ [\![\tau_3]\!]_I) \\
 & \quad\quad ([\![\tau_1\ \subseteq\ \tau_2]\!]_I\ x) \\
 & \quad \text{if } \tau_1\ \subseteq\ \tau_2 \\
[\![\tau_1\ \subseteq\ \tau_2\ \cup\ \tau_3]\!]_I & = \lambda x:\tau_1.\mathsf{Con}(1, \mathsf{Either}\ [\![\tau_2]\!]_I\ [\![\tau_3]\!]_I) \\
 & \quad\quad ([\![\tau_1\ \subseteq\ \tau_3]\!]_I\ x) \\
 & \quad \text{if } \tau_1\ \subseteq\ \tau_3
\end{array}
$$

Figure 11. Auxiliary definitions used to map CUC to CIC

he needs to disallow infinite recursions such as the following one, hinted at in [Coquand 1992]:

$$
\begin{array}{l}
D = \mathsf{Ind}(D:\mathsf{Set})\,\langle((t:\mathsf{Set}) \to t \to t) \to D\rangle\,; \\
d = \mathsf{Con}(0, D)\ \textit{identity}; \\
f = \mathsf{Fix}_0\ f : D \to \bot\ = \\
\quad \lambda d:D.\langle \lambda\_:D.\bot\rangle \mathsf{Case}\ d\ \mathsf{of}\ \langle \lambda id:\_.f\ (id\ D\ d)\rangle; \\
\textit{oops} = f\ d;
\end{array}
$$

5.2 CUC to CIC

We present a translation $[\![\cdot]\!]_I$ which takes any type derivation of CUC and translates it to an equivalent expression in CIC. The complete definition of $[\![\cdot]\!]_I$ can be seen in Figure 12, but the idea is that the elements taken from the common language are kept unchanged and the additional types are mapped as follows:

$$
\begin{array}{ll}
[\![\mathsf{Tuple}_l\ \bullet]\!]_I & = \mathsf{Unit} \\
[\![\mathsf{Tuple}_l\ x:\tau_x, \Delta]\!]_I & = \Sigma x:[\![\tau_x]\!]_I\,.\,[\![\mathsf{Tuple}_l\ \Delta]\!]_I \\
[\![e_1 \equiv e_2]\!]_I & = \mathsf{eq}\ [\![e_1]\!]_I\ [\![e_2]\!]_I \\
[\![\tau_1\ \cup\ \tau_2]\!]_I & = \mathsf{Either}\ [\![\tau_1]\!]_I\ [\![\tau_2]\!]_I \\
\left[\!\left[\mu x:\tau.(\lambda \overrightarrow{z:\tau_z}.e)\right]\!\right]_I & = \mathsf{Ind}(x:[\![\tau]\!]_I) \\
 & \quad\quad \left\langle \overrightarrow{(z:[\![\tau_z]\!]_I)} \to [\![e]\!]_I \to x\ \vec{z} \right\rangle
\end{array}
$$

Where the right hand sides use auxiliary definitions described in Figure 11. Notice also that, as an abuse of notation, we write $[\![e]\!]_I$ instead of $[\![\Gamma \vdash_U e : \tau]\!]_I$.

Contrary to the erasure semantics, in this encoding, cast is not a no-op. Instead, it's the labels on tuples which are ignored. Indeed, since our language distinguishes $\tau_1 \cup \tau_2$ from

$$
\begin{array}{ll}
[\![x]\!]_I & = x \\
[\![\lambda x:\tau_1.e]\!]_I & = \lambda x:[\![\tau_1]\!]_I\,.[\![e]\!]_I \\
[\![e_1\ e_2]\!]_I & = [\![e_1]\!]_I\ [\![e_2]\!]_I \\
[\![(x:\tau_1) \to \tau_2]\!]_I & = (x:[\![\tau_1]\!]_I) \to [\![\tau_2]\!]_I \\
[\![\mathsf{Tuple}_l\ \bullet]\!]_I & = \mathsf{Unit} \\
[\![\mathsf{Tuple}_l\ x:\tau_x, \Delta]\!]_I & = \Sigma x:[\![\tau_x]\!]_I\,.\,[\![\mathsf{Tuple}_l\ \Delta]\!]_I \\
[\![\mathsf{tuple}_l\ \Delta\ \cdot]\!]_I & = \mathsf{unit} \\
[\![\mathsf{tuple}_l\ (x:\tau_\tau, \Delta)\ e, \vec{e}]\!]_I & = \langle x = [\![e]\!]_I\,, \\
 & \quad\quad [\![\mathsf{tuple}_l\ \Delta\ \vec{e}]\!]_I : [\![\mathsf{Tuple}_l\ \Delta]\!]_I\rangle \\
[\![\mathsf{let}\ \cdot = e_1\ \mathsf{in}\ e_2]\!]_I & = [\![e_2]\!]_I \\
[\![\mathsf{let}\ x, \vec{x} = e_1\ \mathsf{in}\ e_2]\!]_I & = \langle \tau_r\rangle \mathsf{Case}\ [\![e_1]\!]_I\ \mathsf{of} \\
 & \quad\quad \left\langle \begin{array}{l} \lambda x:[\![\tau_x]\!]_I\,.\lambda y:[\![\mathsf{Tuple}_l\ \Delta]\!]_I\,. \\ \quad [\![\mathsf{let}\ \vec{x} = y\ \mathsf{in}\ e_2]\!]_I \end{array} \right\rangle \\
 & \quad \text{where } \Gamma \vdash_U e_1 : \mathsf{Tuple}_l\ x:\tau_x, \Delta \\
 & \quad\quad \Gamma, x:\tau_x, \Delta \vdash_U e_2 : \tau_2 \\
 & \quad\quad \tau_r = \lambda x_{e_1} : [\![\mathsf{Tuple}_l\ x:\tau_x, \Delta]\!]_I\,.[\![\tau_2[x_{e_1}.0/x]]\!]_I \\
[\![e_1 \equiv e_2]\!]_I & = \mathsf{eq}\ [\![e_1]\!]_I\ [\![e_2]\!]_I \\
[\![\mathsf{refl}\ e_1]\!]_I & = \mathsf{refl}\ [\![e_1]\!]_I \\
[\![\mathsf{J}\ e_{\equiv}\ e_f\ e]\!]_I & = \mathsf{J}\ [\![e_{\equiv}]\!]_I\ [\![e_f]\!]_I\ [\![e]\!]_I \\
[\![\mu_i x:\tau.e]\!]_I & = \mathsf{Fix}_i\ x : [\![\tau]\!]_I\ =\ [\![e]\!]_I \\
\left[\!\left[\mu x:\tau.(\lambda \overrightarrow{z:\tau_z}.e)\right]\!\right]_I & = \mathsf{Ind}(x:[\![\tau]\!]_I) \\
 & \quad\quad \left\langle \overrightarrow{(z:[\![\tau_z]\!]_I)} \to [\![e]\!]_I \to x\ \vec{z} \right\rangle \\
[\![\mathsf{fold}\ \tau\ \vec{p}\ e]\!]_I & = \mathsf{Con}([\![\tau]\!]_I, 0)\ [\![\vec{p}]\!]_I\ [\![e]\!]_I \\
\left[\!\left[\begin{array}{l}\mathsf{let\ fold}\ \tau\ \vec{p}\ y = e_1 \\ \mathsf{in}\ e_2\end{array}\right]\!\right]_I & = \begin{array}{l} \langle \tau_r\rangle \mathsf{Case}\ [\![e_1]\!]_I\ \mathsf{of} \\ \quad \left\langle \overrightarrow{\lambda z:[\![\tau_z]\!]_I}.\lambda y:\tau_y.[\![e_2]\!]_I \right\rangle \end{array} \\
 & \quad \text{where } \Gamma, \overrightarrow{z:\tau_z}, y:(e_\tau[\tau/x])\ \vec{z} \vdash_U e_2 : \tau_2 \\
 & \quad\quad \tau = \mu x:\_.\lambda \overrightarrow{z:\tau_z}.e_\tau \\
 & \quad\quad \tau_y = [\![e_\tau[\tau/x]]\!]_I \\
 & \quad\quad \tau_r = \lambda \overrightarrow{z:[\![\tau_z]\!]_I}.\lambda y_{e_1}:[\![\tau]\!]_I\ \vec{z}. \\
 & \quad\quad\quad [\![\tau_2[\mathsf{unfold}\ \tau\ \vec{p}\ y_{e_1}/y]]\!]_I \\
[\![\tau_1\ \cup\ \tau_2]\!]_I & = \mathsf{Either}\ [\![\tau_1]\!]_I\ [\![\tau_2]\!]_I \\
[\![\mathsf{cast}\ \tau_1 \subseteq \tau_2\ e]\!]_I & = [\![\tau_1\ \subseteq\ \tau_2]\!]_I\ [\![e]\!]_I \\
\left[\!\left[\begin{array}{l}\mathsf{switch}\ e \\ \mid l\ x\ x_{\equiv} \Rightarrow e_l \\ \mid y\ y_{\equiv} \Rightarrow e_d\end{array}\right]\!\right]_I & = \begin{cases} [\![e_l[e, \mathsf{refl}\ e/x, x_{\equiv}]]\!]_I & \text{if } \tau_d = \bot \\ [\![e_d[e, \mathsf{refl}\ e/y, y_{\equiv}]]\!]_I & \text{if } \tau_l = \bot \\ \langle \tau_r\rangle \mathsf{Case}\ [\![e]\!]_I\ \mathsf{of} \\ \quad \left\langle \lambda x_1:[\![\tau_1]\!]_I\,.e_1,\ \lambda x_2:[\![\tau_2]\!]_I\,.e_2 \right\rangle \\ \quad (\mathsf{refl}\ [\![e]\!]_I) & \end{cases} \\
 & \quad \text{where } \Gamma \vdash_U e : \tau_e \\
 & \quad\quad \tau_e = \tau_1 \cup \tau_2 \ \text{ and } \ \tau_e \overset{l}{\Longrightarrow} \tau_l \cup \tau_d \\
 & \quad\quad \tau_1 \overset{l}{\Longrightarrow} \tau_{l1} \cup \tau_{d1} \text{ and } \tau_2 \overset{l}{\Longrightarrow} \tau_{l2} \cup \tau_{d2} \\
 & \quad\quad e_i = \lambda x_{e\equiv} : [\![e \equiv \mathsf{cast}\ \tau_i \subseteq \tau_e\ x_i]\!]_I\,. \\
 & \quad\quad\quad \left[\!\left[\begin{array}{l}\mathsf{switch}\ x_i \\ \mid l\ x'\ x'_{\equiv} \Rightarrow \\ \quad e_l[\mathsf{cast}\ \tau_{li} \subseteq \tau_l\ x', \mathsf{J}\ x'_{\equiv}\ ?\ e_{\equiv}/x, x_{\equiv}] \\ \mid y'\ y'_{\equiv} \Rightarrow \\ \quad e_d[\mathsf{cast}\ \tau_{di} \subseteq \tau_d\ y', \mathsf{J}\ x'_{\equiv}\ ?\ e_{\equiv}/y, y_{\equiv}]\end{array}\right]\!\right]_I
\end{array}
$$

Figure 12. Mapping CUC to CIC

$\tau_2 \cup \tau_1$, there is a redundancy of information between tuple labels and ordering in union types. In the erasure semantics we used this redundancy to reduce the cost of union types, whereas here we use it to eliminate the need to represent tuple labels.

Lemma 5.1 (Type Preserving translation). *Given* $\Gamma \vdash_U e : \tau$, *we have* $\Gamma \vdash_U \tau : s$ *and* $[\![\Gamma]\!]_I \vdash_I [\![\Gamma \vdash_U e : \tau]\!]_I : [\![\Gamma \vdash_U \tau : s]\!]_I$.

Proof. It's easy to show that given $\Gamma \vdash_U e : \tau$ we have $\Gamma \vdash_U \tau : s$, by induction on the derivation. The remaining part of the proof is longer but is also done by induction on the derivation. It requires proving several auxiliary lemmas such as the fact that a strictly positive recursive type is translated to a strictly positive inductive type, same for terminating recursive functions, as well as the fact that $e \overset{U}{\rightsquigarrow} e'$ implies $[\![e]\!]_I \overset{I}{\rightsquigarrow}^+ [\![e']\!]_I$, which itself requires proving that reduction preserves types. □

Corollary 5.2 (Relative consistency). *Assuming there is no* e *in CIC such that* $\bullet \vdash_I e : \bot$, *then there is no* e *in CUC such that* $\bullet \vdash_U e : \bot$.

Proof. Since $[\![\Gamma \vdash_U \bot : \mathsf{Type}_1]\!]_I = \bot$ and $[\![\bullet]\!]_I = \bullet$, we have that $\bullet \vdash_U e : \bot$ implies $\bullet \vdash_I [\![\Gamma \vdash_U e : \bot]\!]_I : \bot$, so any proof of $\bot$ in CUC can be used to find a proof of $\bot$ in CIC. □

6 Variations and Extensions

The CUC language as presented here was designed so as to keep the type preserving translation to CIC reasonably simple. The ideas can of course be adjusted to different needs. For example, the typing rules of CUC would benefit from being bidirectional, which would let us simplify some of the syntax since things like the Δ arg of "$\mathsf{tuple}_l\ \Delta\ \vec{e}$" could be inferred.

Eliminators One obvious downside of the current presentation is that it still suffers from the fact that an expression that selects the i^{th} field from a tuple cannot have constant size but a size proportional to i. But it is easy to remedy it by making "$e.i$" into a primitive operation rather than mere syntactic sugar. It can even replace "$\mathsf{let}\ \vec{x} = e_1\ \mathsf{in}\ e_2$" altogether. To accompany such change, the termination checker simply needs to be adjusted accordingly, by adding an "$e.i$" rule to the "$v \vdash_U e$ small" judgment.

Similarly, "$\mathsf{unfold}\ \tau\ \vec{p}\ e$" can be made into a primitive operation, or even replace "$\mathsf{let\ fold}\ \tau\ \vec{p}\ x = e_1\ \mathsf{in}\ e_2$" altogether, adding an "$\mathsf{unfold}\ \tau\ \vec{p}\ e$" rule to the "$v \vdash_U e$ small" judgment.

Impredicativity While our calculus is predicative, there is nothing that should prevent adding an impredicative universe to it, just as it is done in CIC: this would simply require adapting the notion of *recursive positions* in a recursive type to constrain the structural induction, as well as disallow strong elimination of large tuples (i.e. tuples that live in Prop but have fields in higher universes).

Of course, the restriction on strong elimination would imply that some projections cannot be defined, so it would make it impossible to do away with the "$\mathsf{let}\ \vec{x} = e_1\ \mathsf{in}\ e_2$" form. The same would likely happen if the tuples were extended with a notion of erasable fields along the lines of the EPTS and ICC systems [Barras and Bernardo 2008; Miquel 2001; Mishra-Linger and Sheard 2008].

6.1 Possible Extensions

When writing proofs, η-equality can be very helpful. For examples, languages like Agda support η-equality rules on functions as well as on records. It is straightforward to add corresponding η-equality rules for functions and records to CUC.

The notion of union type used here is purposefully restrictive, so an obvious extension would be to use a more flexible notion of union types, going in a direction such as the set-theoretic types of Castagna et al. [2016]. Making the union type more flexible could allow the use of extensible sum types and first class cases [Blume et al. 2006].

Rather than fold/unfold primitives, the language could leverage its primitive equality type and simply provide a way to get an equality proof between a μ type and its unfolding, this would make it possible to unfold within an expression (such as unfold all the elements of a list without having to traverse the list).

It is very tempting to try and reify some of the judgments in the type system. For example the judgment "$\tau_1 \subseteq \tau_2$" could be turned into a type, making it possible to cast to a type that is not know at compile time. This could fit very naturally into a system of coercions such as that of Monnier [2007]. Similarly "$\tau \overset{l}{\Longrightarrow} \tau_l \cup \tau_d$" could be defined as a type, and the Δ of "$\mathsf{Tuple}_l\ l\Delta$" could be a *list* of types, so code could abstract over the tail of tuples, like a kind of row polymorphism. Of course, it begs the question of how to define these types without relying on themselves, which might be possible using an approach such as that proposed by Chapman et al. [2010].

7 Related Work

Giménez [1994] was the first effort to align the implementation of inductive types with that of typical algebraic datatypes, thus significantly improving the efficiency of implementation of inductive types.

Odersky and Wadler [1997] implemented algebraic datatypes in the JVM by representing them as (abstract) supertypes where every constructor of the datatype gets a distinct subtype. This results in a very similar representation to the one

used here, where every constructor builds a kind of tuple annotated with a label (the specific subtype corresponding to this constructor) and where our cast operation corresponds to a so-called "up-cast" to the parent type which similarly incurs no runtime cost.

Sulzmann et al. [2007] shows how to reduce generalized algebraic datatypes by compiling them to a core language that instead only provides a notion of proof of equality. We use a similar approach here to encode indexed inductive types, although by working within the context of a dependent language like CC, the equality proofs are easier to manipulate.

Altenkirch et al. [2010] present a calculus on top of which datatypes can be defined. They focus on the simplicity of the metatheory rather than the efficiency of the compiled code: their calculus only offers pairs rather than tuples and has first class tags instead of our untagged unions. It should be straightforward to retrofit our labeled tuples and union types into their calculus while preserving the other properties.

Similarly, Firsov and Stump [2018] shows how to represent datatypes in a language that only provides dependent intersections instead. They share our goal to define datatypes on top of simpler constructs, but they focus on the simplicity of the primitive constructs whereas we are willing to use more complex constructs in exchange for a straightforward efficient compilation.

8 Conclusion

We have presented the Calculus of United Constructions, which is a close cousin of the Calculus of Inductive Constructions, where the inductive type has been carefully split into its constitutive elements in such a way that those lower-level elements do not introduce any extra overhead, making it a good option for an internal representation when compiling a source language similar to CIC.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada grants N^o 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

References

Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. 2010. ΠΣ: Dependent Types without the Sugar. In *International Symposium on Functional and Logic Programming*. 40–55.

Henk P. Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (April 1991), 121–154.

Bruno Barras and Bruno Bernardo. 2008. Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Conference on Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Vol. 4962. Budapest, Hungary.

Matthias Blume, Umut A. Acar, and Wonseok Chae. 2006. Extensible programming with first-class cases. In *International Conference on Functional Programming*. Portland, Oregon, 239–250.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining proofs and programs in a dependently typed language. In *Symposium on Principles of Programming Languages*. ACM Press, 33–45.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyên. 2016. Set-theoretic types for polymorphic variants. In *International Conference on Functional Programming*. 378–391.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *International Conference on Functional Programming*. Baltimore, MD, 3–14.

Thierry Coquand. 1992. Pattern Matching with Dependent Types. In *Workshop on Types for Proofs and Programs*. 66–79.

Denis Firsov and Aaron Stump. 2018. Generic Derivation of Induction for Impredicative Encodings in Cedille. In *Certified Programs and Proofs*. 215–227.

Eduardo Giménez. 1994. *Codifying guarded definitions with recursive schemes*. Technical Report RR1995-07. École Normale Supérieure de Lyon.

Gérard P. Huet, Christine Paulin-Mohring, et al. 2000. The Coq Proof Assistant Reference Manual. Part of the Coq system version 6.3.1.

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Symposium on Principles of Programming Languages*. 177–188.

Xavier Leroy. 1997. The Effectiveness of Type-Based Unboxing. In *International Workshop on Types in Compilation*. BCCS-97-03.

Zhaohui Luo. 1989. ECC, an Extended Calculus of Constructions. In *Annual Symposium on Logic in Computer Science*. 386–395.

Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. 1997. *The Definition of Standard ML Revised*. MIT Press, Cambridge, Massachusetts.

Alexandre Miquel. 2001. The implicit calculus of constructions: extending pure type systems with an intersection type binder and subtyping. In *International conference on Typed Lambda Calculi and Applications*. 344–359.

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Conference on Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Vol. 4962. Budapest, Hungary, 350–364.

Stefan Monnier. 2007. The Swiss Coercion. In *Programming Languages meets Program Verification*. ACM Press, Freiburg, Germany, 33–40.

Stefan Monnier. 2019. Typer: ML boosted with type theory and Scheme. In *Journées Francophones des Langages Applicatifs*. 193–208.

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. Dissertation. Chalmers university.

Martin Odersky and Philip Wadler. 1997. Pizza into Java: Translating Theory into Practice. In *Symposium on Principles of Programming Languages*.

Christine Paulin-Mohring. 1993. Inductive definitions in the system Coq—rules and properties. In *International conference on Typed Lambda Calculi and Applications*, M. Bezem and J. Groote (Eds.). LNCS 664, Springer-Verlag, 328–345.

Zhong Shao. 1997. Flexible Representation Analysis. In *International Conference on Functional Programming*. ACM Press, 85–98.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Types in Language Design and Implementation*. Nice, France, 53–66.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Symposium on Principles of Programming Languages*. ACM Press, 256–270.