Closure Converting the Universes

?? and STEFAN MONNIER, DIRO - Université de Montréal

Type preserving closure conversion of languages with dependent types has proved difficult. It took until 2018 to get the first solution to the problem, and that solution relies on language constructs custom-made for the purpose. A first part of the problem is the fact that closure conversion inevitably requires some form of impredicativity since a function can have free variables that belong to a higher universe than itself, but none of the existing forms of impredicativity (other than those known to be inconsistent) satisfy the needs of closure conversion. A second part is that closure conversion exposes internal details of functions, and those details affect the definitional equality of (converted) functions, hence breaking type preservation.

In this paper we propose to solve the first problem with the use of a new form of impredicativity, and the second with a more restrictive notion of function equality. This allows us to define a closure conversion that relies only on generic language constructs, yet handles a dependently typed language with a tower of universes.

CCS Concepts: • **Theory of computation** → **Type theory**; *Higher order logic*; Logic and verification; • **Software and its engineering** → *Compilers*; Functional languages.

Additional Key Words and Phrases: Closure conversion, Dependent types, Universe polymorphism, Impredicativity, Function equality

1 INTRODUCTION

Closure conversion is a core part of the implementation of a functional programming language. Preserving the full type information across this compilation stage is nowadays common for traditional functional languages, but not so for dependently typed languages, where it stayed an open problem until recently and where the existing solutions are not fully satisfactory yet.

Preserving the types across the various stages of the compiler is particularly important for dependently typed programming languages where the embedded proofs in the source code apply only to ... the source code, whereas for the purpose of formal verification of software we would like to be able to guarantee that those proofs also apply to the compiled code, and that the various pieces that were compiled separately can indeed be linked without breaking any of their invariants.

This tends to get harder the further you progress in the compiler pipeline. And dependent types make it a lot more difficult since runtime terms can appear in the types: as the compilation phases modify those runtime terms, the typing information tends to be affected in profound ways.

In the case of closure conversion, there are three main hurdles: the first is the need to explain to the type system that the closure object we pass at runtime to the closed code indeed contains those precise values that were held in the free variables when we constructed the closure. The second is the fact that closure objects can contain not only other closures of the same type, but also those closures's types, and hence their own type, hence requiring a form of impredicativity. The third is the fact that after closure conversion, the variables captured by a closure are now in full view, which tends to completely change the notion of equality between functions and hence equality between types.

Bowman and Ahmed [2018] provided a first, and only, solution to preserve the types across the closure conversion phase of a dependently typed language, but that solution still has two main shortcomings: first, it handles only the Calculus of Constructions, which is only a subset of most

Authors' address: ??, ??@iro.umontreal.ca; Stefan Monnier, monnier@iro.umontreal.ca, DIRO - Université de Montréal, Montréal, Canada.

2023. 2475-1421/2023/1-ART $15.00
https://doi.org/

dependently typed languages used nowadays, so to be usable for an actual system it would typically need to be extended to cover inductive types and a tower of universes. Second, it relies on a custom construct to represent closures in the output language. While this is a very sensible pragmatic choice, it does beg the questions: what is so special about closures that we do not know how to represent them explicitly without resorting to a custom construct? What extra features would a language need in order to be able to accommodate closure converted code?

In this article, we show an alternate path which makes different tradeoffs in order to find a way to preserve the types across the closure conversion phase while sticking to generic language constructs like dependent tuples and existential packages. Admittedly, our solution still retains a few unusual characteristics, but gets closer to the ideal, and gives a partial answer to the previous questions.

Our contributions are the following:

- A type-preserving closure conversion algorithm for a dependently typed language with a tower of universes.
- A novel solution to the problem of aligning the definition of function equivalence in the code before and after closure conversion.
- A new form of universe polymorphism to handle the need for impredicativity in the closure conversion algorithm.
- The use of equality proofs instead of the translucent types used by Minamide et al. [1996]. This is admittedly already folklore at this point, and was also sketched by Bowman and Ahmed [2018], but we are not aware of any previous work that shows the actual details.

We present the basic problem of type-preserving closure conversion in Section 2. In Section 3 we present each of the three difficulties specific to closure conversion of dependently typed languages, along with how we solved them. In Section 4 we show more formally the input and output languages we use and the closure conversion algorithm. In Section 5 we extend the input language to match the output language. In Section 6 we discuss some of our design decisions.

2 BACKGROUND

Let's consider a predicative Pure Type System (PTS) with a tower of universes as our input language:

$$\begin{array}{llll} \textit{(levels)} & \ell & ::= & 1 \mid S\ \ell \\ \textit{(sorts)} & s & ::= & \mathcal{U}_\ell \\ \textit{(exps)} & e, \tau & ::= & x \mid s \mid (e : \tau) \\ & & & \mid\ (x : \tau_1) \to \tau_2 \mid e_1\ e_2 \mid \lambda x.e \end{array}$$

Here $\mathcal{U}_\ell$ denotes the universe of level ℓ and we use $(x : \tau_1) \to \tau_2$ to denote the type of (dependent) functions, which we will shorten to $\tau_1 \to \tau_2$ when x is not used in τ_2. $(e : \tau)$ is a type annotation to help the bidirectional type checking, so that $\lambda x.e$ does not need a type annotation. 1 is the base universe level and $S\ \ell$ returns the successor of ℓ. Our universe levels start counting at 1 rather than 0 for reasons that will become clear later.

Closure conversion needs to reify closures as data-structures usually represented using tuples, so our output language will additionally require some kind of tuples:

$$\begin{array}{llll} \textit{(telescopes)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\ \textit{(exps)} & e, \tau & ::= & \ldots \\ & & & \mid\ \langle \Gamma \rangle \mid \langle e_1, \ldots, e_n \rangle \mid e.i \end{array}$$

$\langle \Gamma \rangle$ is the type constructor for (dependent) tuples, where Γ lists the types of the fields and where the type of later fields can refer to values of earlier fields; $\langle e_1, \ldots, e_n \rangle$ is the tuple constructor; and $e.i$ returns the i^{th} field of the tuple e. For convenience we will use a bit of syntactic sugar and write

let $\langle x_1, ..., x_n \rangle = e$ in e' to mean $e'[e.1/x_1, ..., e.n/x_n]$ and $\lambda\langle\vec{x}\rangle.e$ to mean $\lambda y.\text{let } \langle\vec{x}\rangle = y \text{ in } e$ where $\vec{x}$ stands for $x_1, ..., x_n$.

Disregarding types for now, the closure conversion of a function like $f = \lambda x.x + y + z$ may look like the following:

$$\begin{aligned}[[f]] \;=\; \langle\; &\langle y, z\rangle, \\ &\lambda\langle x_e, x\rangle.\text{let } \langle y, z\rangle = x_e \text{ in } x + y + z \;\rangle\end{aligned}$$

I.e. a pair whose first element holds the "environment", i.e. the values of the variables captured by the closure (y and z), and whose second element holds the "code", i.e. a closed function. The code in turn expects as argument a pair $\langle x_e, x\rangle$ whose second element (x) is the actual argument to the function, and whose first element (x_e) should be the environment, holding the values of the captured variables, i.e. the first element of the closure.

Accordingly, after closure conversion, a call like $f\ 42$ would turn into:

$$[[f\ 42]] \;=\; \text{let } \langle f_e, f_c\rangle = f \text{ in } f_c\ \langle f_e, 42\rangle$$

If we build the closure naïvely like we did above, its type would look like the following:

$$\begin{aligned}\langle\; &env \;:\; \langle y : \mathsf{Int}, z : \mathsf{Int}\rangle, \\ &code : \langle x_e : \langle y : \mathsf{Int}, z : \mathsf{Int}\rangle, x : \mathsf{Int}\rangle \to \mathsf{Int} \;\rangle\end{aligned}$$

But the type of this pair representing a function whose original type was $\mathsf{Int} \to \mathsf{Int}$ now exposes the number and types of the captured variables. This implies that after closure conversion, two functions which originally had the same type can end up being represented by data structures of incompatible types, thus breaking the type preservation property. For this reason, closures are usually given an existential type that hides the type of the inner tuple representing the environment. Since our tuples are dependently typed, we can do that by prepending to our closures a third field that holds this "hidden" type:

$$\begin{aligned}[[f]] \;=\; &\langle\; \langle y : \mathsf{Int}, z : \mathsf{Int}\rangle, \\ &\quad \langle y, z\rangle, \\ &\quad \lambda\langle x_e, x\rangle.\text{let } \langle y, z\rangle = x_e \text{ in } x + y + z \;\rangle \\ &: \langle\; t \;\;\; : \mathcal{U}, \\ &\quad\; env \;\; : t, \\ &\quad\; code : \langle x_e : t, x : \mathsf{Int}\rangle \to \mathsf{Int} \;\rangle\end{aligned}$$

Now the type does not expose the shape of the captured environment, so two different functions that had the same type before conversion will still have the same type after conversion even if they capture a different number of variables or variables of different types.

This approach works well for System-F [Morrisett et al. 1998], but when we try to apply it to a dependently-typed language there are 3 problems that come up:

(1) Some of the captured variables may also appear in the *type* of the function. In that case, our closure will be ill-typed because the type checker cannot see that the values extracted from x_e are the same as the ones that were captured.
(2) The closure will tend to belong to too high a universe compared to the original function, because it contains the types of the captured variables.
(3) The conversion does not preserve equivalence of terms. For example, when y is equal to 7, the above closure will look very different from the closure generated for the equivalent function $\lambda x.x + 7 + z$. Since terms can appear in types, this means that types may also fail to be equivalent after closure conversion.

All three problems need to be solved if we want the closure conversion of properly typed code to still be properly typed.

3 OUR APPROACH

We present here in more detail the three mentioned problems that afflict type preserving closure conversion in the specific case of a dependently typed language, and we present the solution we used to solve each one.

3.1 Taming dependencies

The first problem we face was identified by Minamide et al. [1996] already: if some of the free variables over which we close a function can appear in its type, then the simple existential encoding fails. For example, say we have the following primitive:

$$\text{makevec} \quad : \quad (t : \mathcal{U}) \to (len : \text{Nat}) \to t \to \text{Vec}\ t\ len$$

and we want to perform closure conversion on the following function:

$$\lambda x.\text{makevec}\ \alpha\ n\ x \quad : \quad \alpha \to \text{Vec}\ \alpha\ n$$

where α and n are its two free variables. The encoding shown before would give us:

$$\begin{array}{l}
[\![\lambda x.\text{makevec}\ \alpha\ n\ x]\!] \\
= \langle\ \langle \alpha : \mathcal{U}, n : \text{Nat} \rangle, \\
\quad \langle \alpha, n \rangle, \\
\quad \lambda \langle x_e, x \rangle.\text{let}\ \langle \alpha', n' \rangle = x_e\ \text{in}\ \text{makevec}\ \alpha'\ n'\ x\ \rangle \\
: \langle\ t \quad : \mathcal{U}, \\
\quad env\ : t, \\
\quad code : \langle x_e : t, x : \alpha \rangle \to \text{Vec}\ \alpha\ n\ \rangle
\end{array}$$

But this code is ill-typed: in makevec $\alpha'\ n'\ x$, we tell makevec that we will provide an element of type α' but then pass it x which has type α. Also, even if we were generous enough to accept the argument x, the return type would not match its expected type because makevec $\alpha'\ n'\ x$ returns a value of type Vec $\alpha'\ n'$ rather than Vec $\alpha\ n$.

In the case of a language like System-F, Morrisett et al. [1998] showed that you can circumvent the problem by not closing over type variables, which are the only variables that can appear in the type in such a language, and since types can be erased we don't really need to close over them. In the above example, maybe α would not be used at run-time and we could then leave it as a free variable, but that is not an option for n since that argument is needed at run time to determine the size of the returned vector.

Arguably the only value that x_e above can take is $\langle \alpha, n \rangle$ and thus α' is always equal to α and n' is always equal to n. You might even prove it via parametricity. Nevertheless, while *we* may know this, the type system does not. Worse, there is simply no way to write a closed function of the above type because in order to return something of type Vec $\alpha\ n$ it would *have to* refer to α and n, defeating the purpose of the closure conversion. For this reason, if we want the code to match its expected type, we need to change the type so as to make it more obvious that we will always receive in x_e the exact value stored in the *env* field of the tuple. Minamide et al. [1996] did this using a feature they call "translucent types", and we could also solve it also using some form of singleton types, but in languages with dependent types, the more natural solution is to use an

equality proof:

$$
\begin{array}{l}
[\![\lambda x.\mathsf{makevec}\ \alpha\ n\ x]\!] \\
= \\
\langle\ \langle \alpha : \mathcal{U}, n : \mathsf{Nat} \rangle, \\
\quad \langle \alpha, n \rangle, \\
\quad \lambda \langle x_e, x, p \rangle . \mathsf{let}\ \langle \alpha', n' \rangle = x_e\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ x' = \mathit{cast}\ (\mathit{eq\_comm}\ p)\ (\lambda x'_e . x'_e.1)\ x\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ \mathit{res} = \mathsf{makevec}\ \alpha'\ n'\ x'\ \mathsf{in} \\
\qquad\qquad\qquad \mathit{cast}\ p\ (\lambda x'_e . \mathsf{Vec}\ (x'_e.1)\ (x'_e.2))\ \mathit{res}\ \rangle \\
: \langle\ t \quad\ : \mathcal{U}, \\
\quad \mathit{env}\ :\ t, \\
\quad \mathit{code} : \langle x_e : t, x : \alpha, p : (x_e = \mathit{env}) \rangle \to \mathsf{Vec}\ \alpha\ n\ \rangle
\end{array}
$$

On the last line we see that the *code* now takes an additional argument p holding a proof that x_e is the same as *env*. In the corresponding code, we see that this proof object is first passed to *cast* in order to turn the input argument x of type α into x' of type $x_e.1$ (which is also known here as α'), and then used a second time at the end to convert the result from Vec $(x_e.1)$ $(x_e.2)$ (also known as Vec α' n') to Vec α n.

The proof p allows us to convert back and forth between the external types which refer to the surrounding variables and the internal types which refer only to variables local to the function.

There is one wrinkle remaining here: the example function we convert is not dependently typed, so the function $\lambda x'_e.\mathsf{Vec}\ (x'_e.1)\ (x'_e.2)$ we pass to the second *cast* to describe the return type does not need to refer to the argument x. In the general case, the return type may refer to the argument x. But this function can't refer to x for two reasons: first, because it would then not be closed, but more importantly because the x it needs would be the actual x on one side of the equality and x' on the other side. The usual solution to this problem is to merge both *cast*s into one as follows:

$$
\begin{array}{l}
\mathsf{let}\ f' = \lambda x'.\mathsf{makevec}\ \alpha'\ n'\ x'\ \mathsf{in} \\
\mathsf{let}\ f = \mathit{cast}\ p\ (\lambda x'_e.(x'_e.1) \to \mathsf{Vec}\ (x'_e.1)\ (x'_e.2))\ f'\ \mathsf{in} \\
f\ x
\end{array}
$$

But we cannot use this solution either because, again, f' is not a closed function. This is a variant of the *convoy pattern* [Chlipala 2013], which always relies crucially on non-closed functions, and it is important to make sure our target language supports it. For this reason, our target language replaces this *cast* operation with a letcast which includes the core element of the convoy pattern. It takes the following form:

$$
\mathsf{letcast}[e_=, e_m]\ x\ =\ e_1\ \mathsf{in}\ e_2 \quad\sim\quad (\mathit{cast}\ e_=\ e_m\ (\lambda x.e_2))\ e_1
$$

The $e_=$ is the proof of equality, e_m is the *motive* which describes how to use the equality to change the types. This letcast operation is arguably the only primitive in our target language that is most visibly adjusted to accommodate the specific needs of a language after closure conversion.

3.2 Taming universes

In the previous section, we just used $\mathcal{U}$ as the universe of types, but in the context of a language with a tower of universes, we need to qualify it with the corresponding level.

Let us consider the source function $g = \lambda x.1 + f\ (x - 1)$, of type $\mathsf{Int} \to \mathsf{Int}$. Its type after closure conversion becomes:

$$\begin{array}{l}[\![\mathsf{Int} \to \mathsf{Int}]\!] \\ = \langle\ t \quad : \mathcal{U}_\ell, \\ \quad\ env\ : t, \\ \quad\ code : \langle x_e : t, x : \mathsf{Int}, p : (x_e = env)\rangle \to \mathsf{Int}\ \rangle\end{array}$$

Where the ℓ subscript in $\mathcal{U}_\ell$ is the universe level inhabited by the captured environment. This tuple type inhabits the universe $\mathcal{U}_{(S\ \ell)}$. But when building the closure for $g = \lambda x.1 + f\ (x - 1)$, the captured environment contains f which is also of type $\mathsf{Int} \to \mathsf{Int}$ and whose type after closure conversion will thus also be the tuple type above. So we would need to fit into the t field of the tuple above a tuple type belonging to $\mathcal{U}_{(S\ \ell)}$, which is clearly too large to fit into the $\mathcal{U}_\ell$ type of this field.

More specifically, we fundamentally need here some form of impredicativity such that our closure's existential quantification can quantify over a universe which includes its own. One solution is to use a language like λ^* that collapses all the universes into a single $\mathcal{U}$ that belongs to itself, but those languages are known to be inconsistent [Hurkens 1995]. All forms of impredicativity known to be consistent are too weak to accommodate our needs. Bowman and Ahmed [2018] were the first to provide a solution to this problem by circumventing it and introducing a custom-made type construct for closures instead of relying on existential quantification.

While their solution only accommodates the Calculus of Constructions, it can likely be adapted to a language with a tower of universes. Yet, we would prefer a solution that does not rely on such custom constructs. So, instead we go back to the tuple type above and see another problem with it: the universe level ℓ needed for the field t of the tuple depends on the types of captured variables, and hence exposes details of the captured variables. So again, we face the problem that the closure conversion of two different closures of the same original type may end up having different types depending on the set of captured variables, thus breaking again our dear type preservation property.

So we apply the same existential quantification trick, but this time quantifying over the universe level:

$$\begin{array}{l}[\![\mathsf{Int} \to \mathsf{Int}]\!] \\ = \exists l.\langle\ t \quad : \mathcal{U}_l, \\ \qquad\ env\ : t, \\ \qquad\ code : \langle x_e : t, x : \mathsf{Int}, p : (x_e = env)\rangle \to \mathsf{Int}\ \rangle\end{array}$$

We don't store the universe level l in the tuple but use a separate $\exists l.\tau$ construct instead for two reasons: first, because manipulating universe levels as first class values is fraught with danger, and second because it lets us make sure that universe levels can be erased, so we do not need to close over them, saving us from a lot of extra complications such as the need to manipulate proofs of equality between universe levels.

With this extra existential quantification, we recover the property that the converted type of a function is always the same regardless of its free variables. But there still remains the question of the universe to which this type should belong. Since it can hold values from arbitrary universe levels, a predicative type theory such as Agda would put such a type in a special universe level ω beyond all other levels and over which $\exists l.\tau$ cannot quantify. This of course would not satisfy our impredicative needs.

A naïve impredicative choice would put this type in the bottom universe instead, but this would immediately lead to an inconsistent type theory because we could then use dummy $\exists l.\tau$ wrappers to bring any type down to the bottom universe, making the language equivalent to λ^*.

So instead we put $\exists l.\tau$ in the universe $\mathcal{U}_{\ell[0/l]}$ where ℓ is the universe level of τ. For the above $[\![\mathsf{Int} \to \mathsf{Int}]\!]$, it results in the universe $\mathcal{U}_1$ since the tuple type itself belong to universe $\mathcal{U}_{(S\ l)}$.

Whether this choice is sound is currently unknown: impredicativity is notoriously tricky and this particular form of impredicativity has not been investigated to any significant extent. The proportion of impredicative systems which have been found to be inconsistent suggests that the odds are not in our favor. This said, we have not been able to adapt known paradoxes like that of Hurkens [1995] to this system, although it might be just a reflection of our inexperience. Our hope is that even if it proves unsound, there might still be a more restrictive version of it which is sound and which at the same time covers the very specific use we make of it: since the l of an object of type $\exists l.\tau$ is actually erased, we cannot extract l out of it, nor can we extract from it any type that belongs to $\mathcal{U}_l$ (such as the field t of our tuple), nor for that matter any value whose type belongs to $\mathcal{U}_l$ (such as the field *env*) or contains a element that belongs to $\mathcal{U}_l$ (such as the field *code*), so really, the only thing we can do with those closure objects is to call them. In this sense, they really are strictly equivalent to the closures they represent from our source language, which is known to be consistent and does not involve any impredicativity.

3.3 Taming function equality

The final remaining problem is the preservation of equality for functions: for the closure conversion to preserve types, we also need to make sure that closure conversion preserves equality between types, i.e. if $\tau_1 \simeq \tau_2$ then $[\![\tau_1]\!] \simeq [\![\tau_2]\!]$.

But this is not the case: in our source language $\lambda x.x + 7$ is equivalent to let $y = 7$ in $\lambda x.x + y$ because let $y = 7$ in $\lambda x.x + y$ can be reduced to $\lambda x.x + 7$. But after closure conversion, these two functions are not equivalent any more. The first will be a closure capturing an empty environment:

$$\begin{array}{l}[\![\lambda x.x + 7]\!] \\ = \langle 1, \langle\langle\bullet\rangle, \langle\rangle, \lambda\langle x_e, x, p\rangle.x + 7\rangle\rangle\end{array}$$

While the second will be a closure capturing an environment containing the value of y:

$$\begin{array}{l}[\![\mathsf{let}\ y = 7\ \mathsf{in}\ \lambda x.x + y]\!] \\ \mathsf{let}\ y = 7\ \mathsf{in}\ \langle 0, \langle\langle y : \mathsf{Int}\rangle, \langle y\rangle, \lambda\langle x_e, x, p\rangle.x + x_e.1\rangle\rangle \\ \rightsquigarrow \\ \langle 1, \langle\langle y : \mathsf{Int}\rangle, \langle 7\rangle, \lambda\langle x_e, x, p\rangle.x + x_e.1\rangle\rangle\end{array}$$

These two closures are clearly different and it seems very difficult to adjust our language's reduction rules so as to allow them to treat those two objects as equivalent. The custom-made closure construct used by Bowman and Ahmed [2018] to circumvent the problem of impredicativity saves them again here, since it allows them to provide a specific η-equivalence rule for those objects. But with our use of tuples, our hands are tied.

We could try to replace the tuple field t with an actual existential quantification (so as to make this t erasable), and then equip this existential quantification with an appropriate η-equivalence rule, as done in [Bowman 2018], but instead we decided to attack this problem at the other end: given the fact that we won't consider those functions equivalent after closure conversion, we change the source language's notion of function equality such that they are also considered as non-equivalent in the source language.

Substitution is not the only operation that threatens the preservation of function equivalence. Another case is β-reductions in the body of a function. For example $\lambda x.x$ and $\lambda x.((\lambda y.x)\ z)$ are traditionally considered as equivalent in type theories, but after closure conversion the former will be a closure capturing no variables while the latter captures a variable z, so they will not be considered equivalent any more.

So we need a new notion of function equality which does not treat those examples as equivalent, yet still allows functions to be equal to themselves. We do that in two parts: first, functions are considered equivalent only if their body is identical (rather than equivalent), basically preventing "reduction under λ"; second we prevent substitutions from entering the body of λ-expressions until the moment they are called, so as to mimic in the source language what happens after closure conversion.

More specifically, we use a limited form of explicit substitutions [Abadi et al. 1990] and change the syntax of functions to $\lambda^{\sigma}x.e$ where σ is a pending substitution (and where we denote the identity substitution by an empty σ). For example, we have:

$$\text{let } y = 7 \text{ in } \lambda x.x + y \quad \simeq \quad \lambda^{7/y}x.x + y$$

While $\lambda x.x + 7$ and $\lambda^{7/y}x.x + y$ behave the same, they are not equivalent any more according to the equivalence relation $\simeq$.

Experience teaches us that confluence of explicit substitutions is not a given [Curien et al. 1996], so we need to define substitutions with care. There are two issues:

- Membership: We can omit the substitution of a variable which is not in the set of free variables. But the set of free variables of a term can be affected by β-reductions, so the sequencing of those operations can affect the end result.
- Ordering: we need to impose some ordering on the elements of a pending substitution to avoid situations where depending on the sequencing of reductions we can end up with either $\lambda^{7/y,3/z}x.x + y + z$ or $\lambda^{3/z,7/y}x.x + y + z$.

While the set of free variables can be different for equivalent terms because of β-reductions, this does not affect λ-expressions of the source language now that we have decided to disallow reductions under λ. So we are free to omit from the pending substitutions those variables which do not appear free in the body of the function without the risk of losing confluence, as long as we make this choice in a deterministic way. We basically have two choices: either never omit them, or always omit them. Omitting them gives us a slightly stronger notion of equivalence but can break the preservation of equivalence of our closure conversion if the closure conversion captures more variables than necessary.

There are also basically two ways to enforce a stable ordering: either place the variables in the pending substitution in the order of their corresponding scope, so a variable cannot "overtake" another, or use an arbitrary ordering defined by the algorithm that collects the set (well: list) of free variables. This latter choice is applicable only if we omit those variables which do not appear freely in the body of the function.

4 CLOSURE CONVERTING THE UNIVERSES

In this section we define the source and target languages for our closure conversion as well as the algorithm itself.

4.1 Source language

The source language we intend to convert has the following syntax:

$$\begin{array}{lrcl}
\textit{(levels)} & \ell & ::= & 1 \mid S\,\ell \\
\textit{(sorts)} & s & ::= & \mathcal{U}_{\ell} \\
\textit{(substitutions)} & \sigma & ::= & \mid \sigma, e/x \\
\textit{(terms)} & e, \tau & ::= & x \mid s \mid (e : \tau) \\
 & & & \mid (x : \tau_1) \to \tau_2 \mid e_1\, e_2 \mid \lambda^{\sigma}x.e
\end{array}$$

$\boxed{\Gamma \vdash e \;\Rightarrow\; \tau}$ and $\boxed{\Gamma \vdash e \;\Leftarrow\; \tau}$: e has type τ in context Γ:

$$\Gamma \vdash \mathcal{U}_\ell \;\Rightarrow\; \mathcal{U}_{(S\,\ell)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x \;\Rightarrow\; \tau}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e \;\Leftarrow\; \tau}{\Gamma \vdash e \;\Leftarrow\; (\tau : s)}\ (\textsc{Strip})$$

$$\frac{\Gamma \vdash \tau \;\Rightarrow\; s \qquad \Gamma \vdash e \;\Leftarrow\; \tau}{\Gamma \vdash (e : \tau) \;\Rightarrow\; \tau}\ (\textsc{Ann}) \qquad \frac{\Gamma \vdash e \;\Rightarrow\; \tau_1 \qquad \tau_1 \simeq \tau_2}{\Gamma \vdash e \;\Leftarrow\; \tau_2}\ (\textsc{Conv})$$

$$\frac{\Gamma \vdash \tau_1 \;\Rightarrow\; \mathcal{U}_{\ell_1} \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 \;\Rightarrow\; \mathcal{U}_{\ell_2}}{\Gamma \vdash (x{:}\tau_1) \to \tau_2 \;\Rightarrow\; \mathcal{U}_{\max(\ell_1,\ell_2)}}\ (\textsc{Pi}) \qquad \frac{\Gamma \vdash e_1 \;\Rightarrow\; (x{:}\tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 \;\Leftarrow\; \tau_1}{\Gamma \vdash e_1\ e_2 \;\Rightarrow\; \tau_2[(e_2{:}\tau_1)/x]}\ (\textsc{App})$$

$$\frac{\Gamma', x{:}\tau_1' \vdash e \;\Leftarrow\; \tau_2' \qquad \Gamma \vdash \sigma \;:\; \Gamma' \qquad \tau_1 = \tau_1'[\sigma] \qquad \tau_2 = \tau_2'[\sigma]}{\Gamma \vdash \lambda^\sigma x.e \;\Leftarrow\; ((x{:}\tau_1) \to \tau_2)}\ (\textsc{Lam})$$

$$\Gamma \vdash \;:\; \Gamma \qquad \frac{\Gamma \vdash \sigma \;:\; \Gamma' \qquad \Gamma \vdash e \;\Rightarrow\; \tau[\sigma]}{\Gamma \vdash \sigma, e/x \;:\; \Gamma', x{:}\tau}$$

$\boxed{e[\sigma]}$: substitution of σ applied to e:

$$\begin{array}{lll}
x[\sigma] & = \begin{cases} \sigma(x) & \text{if } x \in \mathrm{Dom}(\sigma) \\ x & \text{otherwise} \end{cases} & \\
s[\sigma] & = s & \\
(e : \tau)[\sigma] & = (e[\sigma] : \tau[\sigma]) & \\
((x{:}\tau_1) \to \tau_2)[\sigma] & = (x{:}\tau_1[\sigma]) \to (\tau_2[\sigma]) & \\
(e_1\ e_2)[\sigma] & = (e_1[\sigma])\ (e_2[\sigma])) & \\
(\lambda^{\sigma_1} y.e)[\,] & = \lambda^{\sigma_1} y.e & \\
(\lambda^{\sigma_1} y.e)[\sigma_2, e_x/x] & = \begin{cases} (\lambda^{\sigma_1} y.e)[\sigma_2] & \text{if } x \notin \mathrm{fv}(\lambda y.e : (y{:}\tau_1) \to \tau_2) \\ (\lambda^{e_x/x, \sigma_1[e_x/x]} y.e)[\sigma_2] & \text{otherwise} \end{cases} &
\end{array}$$

$\boxed{e_1 \simeq e_2}$: e_1 is equivalent to e_2:

$$(e : \tau) \simeq e \qquad (\lambda^\sigma x.e_1)\ e_2 \simeq e_1[\sigma, e_2/x] \qquad \frac{e \simeq e'}{E[e] \simeq E[e']}\ (\textsc{Cong})$$

$$\begin{array}{lll}
\textit{(subst contexts)} & \Sigma & ::= \Sigma, e/x \mid \sigma, E/x \\
\textit{(eval contexts)} & E & ::= \bullet \mid (E : \tau) \mid (e : E) \\
& & \quad \mid (x{:}E) \to \tau \mid (x{:}\tau) \to E \mid E\ e \mid e\ E \mid \lambda^\Sigma x.e
\end{array}$$

Fig. 1. Typing rules of our source language.

This is the same language as shown at the beginning of Section 2, except for the σ annotation that we added on the λ-expression. The typing rules for this language are shown in Figure 1. We show them in the style of bidirectional rules, so the main judgment is split between $\Gamma \vdash e \;\Rightarrow\; \tau$ and $\Gamma \vdash e \;\Leftarrow\; \tau$, but other than this detail, the main unusual part of those rules comes from the σ annotation on λ-expressions.

$$
\begin{array}{llll}
\textit{(levels)} & \ell & ::= & l \mid 1 \mid S\,\ell \mid \ell_1 \sqcup \ell_2 \\
\textit{(sorts)} & s & ::= & \mathcal{U}_\ell \\
\textit{(ctx)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\textit{(substitutions)} & \sigma & ::= & \mid \sigma, e/x \\
\textit{(terms)} & e, \tau & ::= & x \mid s \mid (e : \tau) \\
 & & & \mid (x : \tau_1) \to \tau_2 \mid e_1\ e_2 \mid \lambda^\sigma x.e \\
 & & & \mid \langle \Gamma \rangle \mid \langle e_1, ..., e_n \rangle \mid e.i \\
 & & & \mid \mathsf{Eq}\ e_1\ e_2 \mid \mathsf{refl} \mid \mathsf{letcast}[e_=, e_m]\ x\ =\ e_1\ \mathsf{in}\ e_2 \\
 & & & \mid \forall l.\tau \mid \Lambda l.e \mid e[\ell] \\
 & & & \mid \exists l.\tau \mid \langle \ell, e \rangle \mid \mathsf{open}\ \langle l, x \rangle = e_1\ \mathsf{in}\ e_2
\end{array}
$$

Fig. 2. Syntax of the target language

A simpler version of the typing rule for $\lambda^\sigma x.e$ could just propagate the substitution σ into the body e and check that $\Gamma, x : \tau_1 \vdash e[\sigma] \iff \tau_2$. Sadly, this would allow e and σ to be ill-typed when considered on their own. So instead the Lam rule enforces that both e and σ are properly typed. The downside is that when σ is not the identity substitution this rule does not have enough information to be decidable. In practice, what this means is that $\lambda^\sigma x.e$ needs to be annotated with its Γ', τ_1', and τ_2' when σ is a non-identity substitution, but those should normally not occur in actual source code, they are only generated internally during reductions.

The figure includes the definition of the substitution operation $e[\sigma]$. While we write it $e[\sigma]$ for convenience, it is actually defined on the typing derivation of e because it requires more typing information than immediately available from the term. This operation shows how a substitution σ_2 applied to a term $\lambda^{\sigma_1} y.e$ is accumulated into the σ annotation instead of continuing its way more eagerly into the body of the function. In that computation we filter the substitution to only keep those variables that are relevant (according to the set of free variables returned by fv), and we keep them in the order they appear in the context.

Finally the figure also includes a definition of the equivalence relation $e_1 \simeq e_2$, where we use evaluation contexts E (and substitution evaluation contexts Σ) to express the set of congruence rules. In type theories, we often omit this part because the primitive rules are presumed to be applicable anywhere, but here we need to clarify that they cannot be used within the body of a function, so the most important element here is the fact that E does *not* include a case of the form $\lambda^\sigma x.E$.

In all other respects, this is a fairly conventional dependently typed λ-calculus with a predicative tower of universes and without universe subsumption.

4.2 Closure conversion

Our target language is a superset of our source language. Its syntax is given in Figure 2. The quantification over universe levels adds to our levels ℓ two new cases, one for level variables l, and another for the maximum of two levels $\ell_1 \sqcup \ell_2$. The sorts, contexts, and substitutions stay unchanged. But we add many new elements to the terms:

- $\langle \Gamma \rangle$ is the type of dependent tuples where Γ describes the fields and their types; $\langle e_1, ..., e_n \rangle$ is the corresponding term constructor; and $e.i$ is the elimination form which projects the i^{th} field out of e.
- $\mathsf{Eq}\ e_1\ e_2$ is the type of proofs that e_1 and e_2 are equal; refl is the corresponding constructor of the proof by reflexivity; and $\mathsf{letcast}[e_=, e_m]\ x\ =\ e_1\ \mathsf{in}\ e_2$ is the elimination form which

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![s]\!] & = & s \\
[\![(e : \tau)]\!] & = & ([\![e]\!] : [\![\tau]\!]) \\
[\![(x{:}\tau_1) \to \tau_2]\!] & = & \exists l.\langle\, t \quad : \mathcal{U}_l, \\
 & & \qquad env \;:\; t, \\
 & & \qquad code :\; \langle x_e : t, x : [\![\tau_1]\!], p : (x_e = env)\rangle \to [\![\tau_2]\!] \,\rangle \\
[\![e_1\ e_2]\!] & = & ((\text{open } \langle l, c\rangle = [\![e_1]\!] \text{ in } c.3\ \langle c.2, [\![e_2]\!], \text{refl}\rangle) : [\![\tau]\!]) \\
 & & \text{where } \Gamma \vdash e_1\ e_2 \;\Rightarrow\; \tau \\
[\![\lambda^\sigma x.e]\!] & = & \langle \ell, \langle [\![\Gamma']\!]\rangle, \langle [\![\vec{x}[\sigma]]\!]\rangle, \lambda\langle x_e, x', x_=\rangle.body\rangle\rangle \\
 & & \text{where } \Gamma, x{:}\tau_x \vdash e \;\Leftarrow\; \tau_r \\
 & & \qquad \vec{x_f} \;=\; \mathsf{fv}(\lambda x.e : (x{:}\tau_x) \to \tau_r) \\
 & & \qquad \Gamma' \;=\; \Gamma|_{\vec{x_f}} \;=\; \vec{x} : \vec{\tau} \\
 & & \qquad \Gamma \vdash \langle \Gamma'\rangle \;\Rightarrow\; \mathcal{U}_\ell \\
 & & \qquad f_m \;=\; \lambda\langle\vec{x}\rangle.(x{:}[\![\tau_x]\!]) \to [\![\tau_r]\!] \\
 & & \qquad body = \; \mathsf{let}\ \langle\vec{x}\rangle = x_e\ \mathsf{in} \\
 & & \qquad\qquad \mathsf{letcast}[x_=, f_m]\ x \;=\; x'\ \mathsf{in}\ [\![e]\!]
\end{array}
$$

Fig. 3. The closure conversion itself

takes a proof $e_=$: Eq e_i e_o, a motive e_m, and returns let $x = e_1$ in e_2 except that e_2 is typed in a context where the e_o are replaced by e_i. Other than this type-twist, it is expected to be compiled to the same code as let $x = e_1$ in e_2.

- $\forall l.\tau$ is the type of code that is polymorphic over universe level l; $\Lambda l.e$ is the corresponding introduction form; and $e[\ell]$ the corresponding elimination form which instantiates the level variable with level ℓ.
- $\exists l.\tau$ is the existential type that quantifies over universe level l; $\langle \ell, e\rangle$ is the corresponding introduction form; and open $\langle l, x\rangle = e_1$ in e_2 is its elimination form.

We can now show the actual closure conversion itself, denoted $[\![\cdot]\!]$, which is in Figure 3. An important detail to note in that figure is an abuse of notation: while we write $[\![e]\!]$, the conversion algorithm does not take a mere term e as argument but it really operates on a typing derivation of e because it needs more type information than is readily provided in e itself. We use this notational abuse in the hope to make the code more readable. In contrast, $[\![\cdot]\!]$ does return a mere term and not a full typing derivation. While we like to think of it as a conversion from intrinsically typed terms to intrinsically typed terms, we prefer to return a mere term so that we can separately state and prove that it does indeed preserve typing.

The first three cases are of no interest. The first interesting case is the one for $(x{:}\tau_1) \to \tau_2$ where we state the type of a closure to be fundamentally a 4-tuple made of (in reverse order) a closed function we denote as *code*, a captured environment *env*, its type t, and its universe level l.

The case for $e_1\ e_2$ takes such a closure object and calls it: it uses open to get to the universe level l and the rest of the closure c. At that point $c.3$ holds the actual code and $c.2$ is the captured environment. The proof that $x_e = env$ is trivially refl since at this point both x_e and env have been replaced by $c.2$.

The more intricate case is for $\lambda^\sigma x.e$: there we actually build the closure object, made of its code $\lambda\langle x_e, x', x_=\rangle.body$, its captured environment $\langle [\![\vec{x}[\sigma]]\!]\rangle$, its type $\langle [\![\Gamma']\!]\rangle$, and its universe level ℓ. By $[\![\vec{x}[\sigma]]\!]$ and $[\![\Gamma']\!]$ we mean to apply $[\![\cdot]\!]$ and $\cdot[\sigma]$ pointwise to, respectively, $\vec{x}$ and Γ'. To build

$$\begin{array}{lrcl} \textit{(level contexts)} & L & ::= & \bullet \mid S\ L \mid L \sqcup \ell \mid \ell \sqcup L \\ \textit{(eval contexts)} & E & ::= & \ldots \mid \mathcal{U}_L \end{array}$$

$$\frac{\Gamma \vdash \tau_1 \ \Rightarrow\ \mathcal{U}_{\ell_1} \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 \ \Rightarrow\ \mathcal{U}_{\ell_2}}{\Gamma \vdash (x{:}\tau_1) \to \tau_2 \ \Rightarrow\ \mathcal{U}_{(l_1 \sqcup \ell_2)}}\ (\textsc{Pi})$$

$$0 \sqcup \ell \simeq \ell \qquad\qquad \ell \sqcup 0 \simeq \ell \qquad\qquad (S\ \ell_1) \sqcup (S\ l_2) \simeq S\ (l_1 \sqcup l_2)$$

Fig. 4. Differences with source language for the core target language.

$$\Gamma \vdash \langle\bullet\rangle \ \Rightarrow\ \mathcal{U}_1 \qquad\qquad \frac{\Gamma \vdash \langle\Gamma'\rangle \ \Rightarrow\ \mathcal{U}_{\ell_1} \qquad \Gamma, \Gamma' \vdash \tau \ \Rightarrow\ \mathcal{U}_{\ell_2}}{\Gamma \vdash \langle\Gamma', x{:}\tau\rangle \ \Rightarrow\ \mathcal{U}_{\ell_1 \sqcup \ell_2}}\ (\Sigma)$$

$$\Gamma \vdash \langle\rangle \ \Rightarrow\ \langle\bullet\rangle \qquad\qquad \frac{\Gamma \vdash \langle\vec{e}\rangle \ \Leftarrow\ \langle\Gamma'\rangle \qquad \Gamma' = \vec{x} : \vec{\tau} \qquad \Gamma \vdash e \ \Leftarrow\ \tau[\vec{e}/\vec{x}]}{\Gamma \vdash \langle\vec{e}, e\rangle \ \Leftarrow\ \langle\Gamma', x{:}\tau\rangle}\ (\textsc{Tup})$$

$$\frac{\Gamma \vdash e \ \Rightarrow\ \langle\Gamma'\rangle \qquad \Gamma' = \vec{x} : \vec{\tau}}{\Gamma \vdash e.i \ \Rightarrow\ \tau_i[e.1/x_1, \ldots, e.(i-1)/x_{i-i}]}\ (\textsc{Proj})$$

$$\langle\vec{e}\rangle.i \simeq e_i$$

Fig. 5. Typing rules for dependent tuples.

those, we first need to extract the type information of e (provided by the typing derivation we receive as argument), then we compute the *set* of free variables $\vec{x_f}$ and from that we compute Γ' to get the *list* of free variables, placed in the same order they appear in the context so as to preserve any type dependencies between them. We then compute the type of $\langle\Gamma'\rangle$ to find the universe level ℓ in which it lives. f_m is the motive of the letcast we use later, which describes how to redirect variable references between x_e and *env*. Finally the *body* just unpacks the environment into the corresponding variables and evaluates $[\![e]\!]$, while carefully adjusting the types of x and of the result to redirect references from the types to variables captured by the function.

4.3 Target language

We have already shown the syntax of the target language, but we present here its actual definition in the form of its typing and conversion rules.

Since our target language is a superset of our source language, the core elements are the same and basically share the same rules, except for changes to the universe levels. Figure 4 shows the parts of the rules that changed, fundamentally due to the fact that the $\max(\ell_1, \ell_2)$ computation that used to be performed at the metalevel is now internalized as $l_1 \sqcup \ell_2$, which in turn requires new conversion rules to define the semantics of this operation.

Figure 5 shows the rules governing the dependent tuples, which are completely standard. The only significant complexity is in the rules Proj and Tup which need to take into account the possible

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau \qquad \Gamma \vdash \tau \Rightarrow s}{\Gamma \vdash \mathsf{Eq}\ e_1\ e_2 \Rightarrow s}\ (\textsc{Eq}) \qquad \frac{e_1 \simeq e_2}{\Gamma \vdash \mathsf{refl} \Leftarrow \mathsf{Eq}\ e_1\ e_2}\ (\textsc{Refl})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_= \Rightarrow \mathsf{Eq}\ e_1\ e_2 \qquad \Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_m \Leftarrow (x:\tau) \to s \\ e_m\ e_1 = (x:\tau_{a1}) \to \tau_{r1} \qquad e_m\ e_2 = (x:\tau_{a2}) \to \tau_{r2} \qquad \Gamma \vdash e_a : \tau_{a2} \qquad \Gamma, x:\tau_{a1} \vdash e_r : \tau_{r1}\end{array}}{\Gamma \vdash \mathsf{letcast}[e_=, e_m]\ x = e_a\ \mathsf{in}\ e_r \Rightarrow \tau_{r2}[e_a/x]}$$

$$\mathsf{letcast}[\mathsf{refl}, e_m]\ x = e_a\ \mathsf{in}\ e_r \simeq e_r[e_a/x]$$

Fig. 6. Typing rules for the identity type.

$$\frac{\Gamma \vdash \tau \Rightarrow \mathcal{U}_\ell}{\Gamma \vdash \forall l.\tau \Rightarrow \mathcal{U}_{\ell[0/l]}}\ (\text{U-}\forall) \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash \Lambda l.e \Leftarrow \forall l.\tau}\ (\text{U-}\Lambda) \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e[\ell] \Rightarrow \tau[\ell/l]}\ (\text{U-}\textsc{App})$$

$$\frac{\Gamma \vdash \tau \Rightarrow \mathcal{U}_\ell}{\Gamma \vdash \exists l.\tau \Rightarrow \mathcal{U}_{\ell[0/l]}}\ (\text{U-}\exists) \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \langle \ell, e \rangle \Rightarrow \exists l.\tau}\ (\text{U-}\textsc{Pack})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \exists l.\tau_1 \qquad l' \text{ is fresh} \qquad \Gamma, x:\tau_1[l'/l] \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{open}\ \langle l, x \rangle = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau}\ (\text{U-}\textsc{Open})$$

$$(\Lambda l.e)[\ell] \simeq e[\ell/l] \qquad \mathsf{open}\ \langle l, x \rangle = \langle \ell, e_1 \rangle\ \mathsf{in}\ e_2 \simeq e_2[\ell/l][e_1/x]$$

Fig. 7. Typing rules for universe polymorphism.

dependencies between the fields, which requires applying substitutions to replace references to previous fields with those fields' values when returning the type of fields.

Figure 6 shows the rules governing the equality (also called identity) type. The Eq and Refl rules are standard, and while the letcast primitive is non-standard due to the need to accommodate the constraint that we want it to be able to write our code such that all λ-expressions are closed, it is more verbose than complex: its rule simply encodes what a standard elimination rule like J would provide when combined with the convoy pattern.

Figure 7 shows the rules that govern universe polymorphism and the associated existential quantification of universe level. Note that our closure conversion algorithm uses only the $\exists$ quantification on universe levels, but we include the $\forall$ quantification anyway since it is very closely tied and corresponds to a more common feature in proof assistants.

Rules U-Λ, U-App, E-Pack, and U-Open for the introduction and elimination forms of $\forall$ and $\exists$ as well as the corresponding conversion rules are reasonably standard in the sense that they are almost identical to the corresponding rules of System-F, except for the fact that they quantify over universe levels rather than over types. Indeed, we even chose a presentation where we do not keep track of the set of universe level variables in scope, as is done in some presentations of System-F. This was done to avoid having to carry around another context in all the typing rules, so as to make the rules easier on the eyes, but it is otherwise of no particular significance.

An important detail to note here is that the substitutions $e[\ell/l]$ are different from the substitutions $e[\sigma]$: universe level variables are treated completely normally with the usual substitution rules, which means that they propagate into the body of λ expressions unlike substitutions for term variables x.

The novelty here is rather in the rules U-$\forall$ and U-$\exists$ which determine the universe level in which those universe-polymorphic terms live. These are the rules which introduce impredicativity in our language.

4.3.1 The base universe question. A careful comparison of the syntax of our rules shows that our treatment of the base universe is not quite standard: while our language syntax uses 1 as the base universe level and our unit type $\langle\bullet\rangle$ indeed lives in this universe, our universe rules in Figure 4 as well as rules U-$\forall$ and U-$\exists$ use 0 as the base universe level instead. These are not inconsistencies. Indeed, we use 0 as a kind of basement, or arguably more like a *crawl space* because no type lives in this universe and the source code cannot access it. It is only used internally in U-$\forall$ and U-$\exists$.

In a predicative PTS with a tower of universes, the base universe is always empty, at least until one explicitly extends the language by adding elements to it, e.g. by adding a unit type. For the same reason, our U-$\forall$ and U-$\exists$ rules can never place a type in universe $\mathcal{U}_0$: the stratification of our universes means that if l appears somewhere in τ, τ itself necessarily lives in a universe level strictly higher than l, so $\ell[0/l]$ cannot be 0. This makes universe $\mathcal{U}_0$ annoyingly special. For example it means our encoding of closures can only represent closures in universes $\mathcal{U}_1$ or higher. So we decided to make all type constructors "equal" by keeping $\mathcal{U}_0$ empty, moving the unit type to $\mathcal{U}_1$ and preventing the source code from referring to $\mathcal{U}_0$. The only use for universe level 0 in our system is to make it possible for U-$\forall$ and U-$\exists$ to place their types in the "real" bottom universe $\mathcal{U}_1$.

4.4 Properties

Our closure conversion algorithm is designed like a syntactic model, following the approach advocated in [**?**].

Note: since our conversion takes an actual typing derivation as input, we cannot really manipulate "naked" terms, and thus all the properties and proofs below require that we first redefine our conversion relation $e_1 \simeq e_2$ to be a typed conversion relation $\Gamma \vdash e_1 \simeq e_2 \ : \ \tau$. This is a boring and mechanical process which result in significantly more verbose rules, so we preferred to present the untyped rules which focus on the important elements. Sadly, this also forces the three lemmas below to be mutually recursive.

Lemma 4.1 (Substitution lemma).
If e_1 and e_2 are expressions in our source language and $\Gamma, x : \tau_2 \vdash e_1 \ \Leftarrow \ \tau_1$ and $\Gamma \vdash e_2 \ \Leftarrow \ \tau_2$ then $[\![e_1[e_2/x]]\!] \simeq [\![e_1]\!]\,[[\![e_2]\!]/x]$.

Proof. By induction on e_1. □

Lemma 4.2 (Computational soundness).
If e_1 and e_2 are two expressions in our source language, and $e_1 \simeq e_2$, and $\Gamma \vdash e_1 \ \Leftarrow \ \tau$ and $\Gamma \vdash e_2 \ \Leftarrow \ \tau$, then $[\![e_1]\!] \simeq [\![e_2]\!]$.

Proof. By induction on the derivation of $e_1 \simeq e_2$. The only non-trivial case is the β-reduction rule, where it is easy to see that the various introduction forms used by the encoding of the λ-expression are all canceled by the corresponding elimination forms of the encoding of the application. □

Lemma 4.3 (Typing soundness).
If e is an expression in our source language and $\Gamma \vdash e \ \Leftarrow \ \tau$, then $[\![\Gamma]\!] \vdash [\![e]\!] \ \Leftarrow \ [\![\tau]\!]$.

Proof. This first requires proving that the premises imply that $\Gamma \vdash \tau \implies s$, without which one cannot talk about $[\![\tau]\!]$. This is a trivial side lemma proved by induction on the typing derivation. The rest of the proof is by induction on the typing derivation. It is rather tedious and follows the same general structure as the usual proof of type preservation of closure conversion, such as found in [?]. □

Lemma 4.4 (Conservation of freedom).
If e is an expression in our source language and $\Gamma \vdash e \Leftarrow \tau$, then $fv(e : \tau) = fv([\![e]\!] : [\![\tau]\!])$.

Proof. By induction on the typing derivation. □

Lemma 4.5 (Closedness).
If e is an expression in our source language and $\Gamma \vdash e \Leftarrow \tau$, then for all λ-expressions $\lambda^\sigma x.e$ in $[\![e]\!]$ we have that $fv(\lambda^\sigma x.e) = \emptyset$.

Proof. By induction on the typing derivation. We can see that all the $\lambda^\sigma x.e$ that can occur in the converted code come from the conversion of a $\lambda^\sigma x.e$ in the input. Furthermore, we can see that for such an input we generate two λ-expressions, both of which are closed by construction, thanks to the conservation of freedom lemma. □

5 CLOSURE CONVERTING A BIGGER LANGUAGE

Our source language was purposefully very limited, so that we could focus on the important details, but of course we want to be able to scale this to a more realistic language. Luckily, this source language also hit the most problematic spots of closure conversion, so it is fairly straightforward to extend our result to a more general source language.

To get started, we can extend our source language with (dependent) tuples, using the same syntax and rules as we used in our target language. Extending the conversion function to handle these constructs is simply:

$$\begin{array}{lcl} [\![\langle \Gamma \rangle]\!] & = & \langle [\![\Gamma]\!] \rangle \\ [\![\langle \vec{e} \rangle]\!] & = & \langle [\![\vec{e}]\!] \rangle \\ [\![e.i]\!] & = & [\![e]\!].i \end{array}$$

Next step, we can extend our source language with the same $\exists$ and $\forall$ quantification over universe levels as we have in our target language. Universe levels are second class citizens which can be erased, just like types in System-F, so our closures do not need to close over universe level variables, which means we can use the same simple approach as was used in [Morrisett et al. 1998]:

$$\begin{array}{lcl} [\![\forall l.\tau]\!] & = & \forall l.\, [\![\tau]\!] \\ [\![\Lambda l.e]\!] & = & \Lambda l.\, [\![e]\!] \\ [\![e[\ell]]\!] & = & [\![e]\!]\, [\ell] \\ [\![\exists l.\tau]\!] & = & \exists l.\, [\![\tau]\!] \\ [\![\langle \ell, e \rangle]\!] & = & \langle \ell, [\![e]\!] \rangle \\ [\![\mathsf{open}\ \langle l, x \rangle = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{open}\ \langle l, x \rangle = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \end{array}$$

Not shown here: in order for this to work correctly, one has to be careful to define $\mathsf{fv}(e)$ such that it only considers term variables x and ignores universe level variables l.

Finally, we can try to extend our source language to be the same as our target language by adding the remaining equality type constructs. But this hits a minor hurdle: when trying to convert letcast expressions, we find that we cannot do that because it only accepts a λ-expression for its e_m motive and we can neither guarantee that this λ-expression is always closed nor convert it into something closed when it is not.

$$\frac{\begin{array}{c}\Gamma \vdash e_= \;\Rightarrow\; \mathsf{Eq}\ e_1\ e_2 \qquad \Gamma \vdash e_1 \;\Rightarrow\; \tau \qquad \Gamma \vdash e_c \;\Rightarrow\; \tau_c \\ \Gamma \vdash e_m \;\Leftarrow\; (x{:}\langle \tau_c, \tau\rangle) \to s \qquad e_m\ \langle e_c, e_1\rangle = (x{:}\tau_{a1}) \to \tau_{r1} \qquad e_m\ \langle e_c, e_2\rangle = (x{:}\tau_{a2}) \to \tau_{r2} \\ \Gamma \vdash e_a \;:\; \tau_{a2} \qquad \Gamma, x{:}\tau_{a1} \vdash e_r \;:\; \tau_{r1}\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{letcast}[e_=, e_c, e_m]\ x \;=\; e_a\ \mathsf{in}\ e_r \;\Rightarrow\; \tau_{r2}[e_a/x] \\ \mathsf{letcast}[\mathsf{refl}, e_c, e_m]\ x \;=\; e_a\ \mathsf{in}\ e_r \quad\simeq\quad e_r[e_a/x]\end{array}}$$

Fig. 8. Refinement of letcast with e_c

To solve this problem, we take a page from traditional C programming conventions where similar problems are solved by making higher-order functions take an additional `void*` argument that they just pass on to their first-class function argument. Concretely we add an e_c argument to letcast which is simply passed on to the e_m function.

$$\mathsf{letcast}[e_=, e_c, e_m]\ x \;=\; e_1\ \mathsf{in}\ e_2 \quad\sim\quad (\mathit{cast}\ e_= \ (\lambda y.e_m\ \langle e_c, y\rangle)\ (\lambda x.e_2))\ e_1$$

Figure 8 shows how to adjust the typing and equivalence rules for the new construct. It is easy to adjust our closure conversion to use this new letcast: since do not need e_c when converting a λ-expression, we can just pass a dummy unit type argument for it.

With this wrinkle adjusted, we can now define the closure conversion of the equality type constructs as well:

$$\begin{array}{lcl}[\![\mathsf{Eq}\ e_1\ e_2]\!] & = & \mathsf{Eq}\ [\![e_1]\!]\ [\![e_2]\!] \\ [\![\mathsf{refl}]\!] & = & \mathsf{refl} \\ [\![\mathsf{letcast}[e_=, e_c, e_m]\ x \;=\; e_1\ \mathsf{in}\ e_2\ ]\!] & = & \mathsf{letcast}[[\![e_=]\!], e'_c, e'_m]\ x \;=\; [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \\ & & \quad \mathsf{where}\ \ \vec{x_f} \;=\; \mathsf{fv}(e_m) \\ & & \qquad\qquad e'_c \;=\; \langle \vec{x_f}, [\![e_c]\!]\rangle \\ & & \qquad\qquad e'_m \;=\; \lambda\langle\langle \vec{x_f}, x_c\rangle, x_m\rangle.\ [\![e_m\ \langle x_c, x_m\rangle]\!]\end{array}$$

With this extension, our closure conversion algorithm accepts the same input language as its output language.

6 DISCUSSION

Our aim was to cover a fairly realistic source language and to use a target language that is as "generic" as we can. Here we discuss several design decisions as well as the limits of our current work.

6.1 Source language features

While our source language does not cover all of the features of a real language like Idris or Agda, we do cover a significant part. The main missing functionality would be things like inductive types, coinductive types, a Prop universe, erasable arguments, and linearity.

Adding support for inductive types should not be problematic: we already support dependent tuples and equality types, so fundamentally all that is missing is sum types and recursive types. We do not foresee any particular difficulty here, as long s the usual syntactic checks termination checks are sufficiently refined not to be confused by the layers of tuples and letcast added by the closure conversion. The same should hold for coinductive types.

There will probably need to be a need for some adjustments to some constructs, like we had to do with letcast. E.g. Coq's elimination principle for inductive types also takes a motive to describe the dependent output type and that annotation has to be a function rather than a closure, just like the e_m of letcast. This could be solved in the same way we did for letcast.

In contrast, it is very much unclear how an impredicative Prop universe would interact with the rest of this language. It is also unclear how the impredicativity already provided by our language compares to that of Prop.

Adding support for erasable arguments and linearity seems to fall in-between. It is not immediately obvious, but it might be feasible: adding those features to the language itself should not pose any specific difficulty; the main difficulty would be performing closure conversion without changing the erasability/linearity of variables.

6.2 Efficiency

Performing closure conversion correctly is a good first step, but generating efficient code is also important. The current closure conversion is not fully satisfactory in this regard and solving some of those issues may not be straightforward.

The main issue with our conversion is that our closures are represented as triples of the code, the environment, and its type (the universe level can be assumed to be erased by later phases). We may be able erase also the type of the environment, leaving us with a pair of the code and the environment. While this is the "official" definition of a closure, in practice closures are more often implemented by merging the inner tuple representing the environment with the outer pair, resulting in a tuple that contains the code in one field and the free variables in the other fields. Worse: it also the code receives as argument not just the environment, but the whole closure (since they do not exist separately any more), which requires makes the code's type recursive. It is unclear how to allow this kind of recursion without breaking the logic's soundness.

Another related aspect is that it is common to place the code in the first field and the captured variables afterwards, whereas our encoding requires fundamentally the environment to come first because the type of the code must refer to the environment. Dependent type system always assume a kind of left-to-right ordering of dependencies, but sometimes practical concerns may require fields to be layed out differently. At the lower level these ordering issues should not matter, so it would be good to find a way to encode dependencies separately from the ordering, but how to allow that without breaking the logic is again unclear.

6.3 Function equality

One of the most unusual aspect of our language is the way it defines function equality. This clearly weakens the definitional equality of our logic, and to make matters worse it makes our language significantly more complex.

An naive counter argument might point out that the definitional equality between functions in languages like Agda and Coq are already too weak anyway, so it is very commonplace to supplement with the functional extensionality axiom. Sadly, this breaks our closure conversion: this axiom applies to λ-expressions and not to the corresponding closure objects after conversion, for the very same reason that pushed us to weaken our definitional equality.

Clearly, this is the most problematic part of our approach. We hope to revise it in a future work by strengthening our target language with the use of quotient types so that it can faithfully represent the full definitional equality of source λ-expression after they're encoded as closures. This should also be amenable to an extension that handles the functional extensionality axiom.

There might still be a potential other use for our functions' weak definitional equality: this notion of equality seems to be compatible with a notion of runtime equality testing, although it would impose a few restrictions on the kinds of optimizations that the compiler can safely apply (for example, the η rule is not valid in general). More specifically it could be used to provide a precise semantics for runtime equality tests between functions, thus solving in a different way the problem

that arguably lead the SML designers to introduce the unsatisfactory notion of eqtype [**?**], and arguably motivated the invention of type classes.

6.4 Universe polymorphism

While our minimal source language is known to be sound, the logical consistency of our target language is a big question mark because of its reliance on a novel notion of impredicativity introduced by more aggressive rules for universe polymorphism.

In the specific way the $\exists$ quantification is used here, the rules proposed seem eminently reasonable: they place the closure objects right in the exact same universe level that they original λ-expression occupied in the source code. Sadly, that does not guarantee that they are sound in general.

Our intuition as for why they *may* not be completely crazy is that the second-class status of universe levels makes universe polymorphic definitions enjoy a strong for of parametricity. Agda's position says that $\forall l.\tau$ can be modeled as a set theoretic function which for every level l returns the corresponding τ, so this function is clearly very large since it includes all the possible τ one can get for all the possible levels with which we can instantiate it. For this reason, if $\Gamma \vdash \tau : \mathcal{U}_\ell$ Agda places $\forall l.\tau$ in the universe $\sup_l \ell$ which they represent as ω. Our typing rules basically take the opposite position, considering that the type $\forall l.\tau$ is arguably smaller than any given instantiation of τ since it only holds those functions which can be used at any universe level (just like the type $\forall t.t \longrightarrow t$ is so small that it only contains a single element), so it places it in the universe $\inf_l \ell$, hence $\ell[0/l]$.

We do not know that this intuition holds water, sadly. But we can't see how to type closure converted code of a source language with a tower of universes without using something similar to the rules we propose: in a sense, the rules we use arise naturally in our encoding. Of course, there is always the emergency escape hatch of resorting to custom-made constructs like the one used in [Bowman and Ahmed 2018].

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada grant N^o 298311/2012 and RGPIN-2018-06225 as well as by the Fonds de Recherche du Québec - Nature et Technologie grant N^o 253521. 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 or the FRQNT.

REFERENCES

Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1990. Explicit Substitutions. In *Symposium on Principles of Programming Languages*. ACM Press, 31–46.

Bowman. 2018. *Compiling with Dependent Types*. Ph. D. Dissertation. Northeastern University. https://williamjbowman.com/resources/wjb-dissertation.pdf

William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Programming Languages Design and Implementation*. 797–811. https://doi.org/10.1145/3192366.3192372

Adam Chlipala. 2013. *Certified Programming with Dependent Types*. MIT Press.

Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. 1996. Confluence properties of Weak and Strong Calculi of Explicit Substitutions. *Journal of the ACM* 43, 2 (1996), 362–397.

Antonius Hurkens. 1995. A simplification of Girard's paradox. In *International conference on Typed Lambda Calculi and Applications*. 266–278. https://doi.org/10.1007/BFb0014058

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Symposium on Principles of Programming Languages*. ACM Press, 271–283. https://doi.org/10.1145/237721.237791

Stefan Monnier and Nathaniel Bos. 2019. Is Impredicativity Implicitly Implicit?. In *Types for Proofs and Programs (Leibniz International Proceedings in Informatics (LIPIcs))*. 9:1–9:19. https://doi.org/10.4230/LIPIcs.TYPES.2019.9

Stefan Monnier and David Haguenauer. 2010. Singleton types here, Singleton types there, Singleton types everywhere. In *Programming Languages meets Program Verification*. ACM Press, 1–8. https://doi.org/10.1145/1707790.1707792

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *Symposium on Principles of Programming Languages*. 85–97. https://doi.org/10.1145/319301.319345

Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 2002. A Type System for Certified Binaries. In *Symposium on Principles of Programming Languages*. 217–232. https://doi.org/10.1145/503272.503293

Vladimir Voevodsky. 2012. A universe polymorphic type system. https://www.math.ias.edu/~dgrayson/Voevodsky-old-files/files/files-annotated/Dropbox/Unfinished_papers/Type_systems/UPTS_current/Universe_polymorphic_type_sytem.pdf