Closure Converting the Universes

ANONYMOUS AUTHOR(S)

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 and does not support the customary tower of universes. There are basically three sources of difficulty. The first of them occurs when we need to close over variables which appear also in the type of the function, and can be solved with singleton types or translucent types. The second difficulty is the fact that closure conversion inevitably requires some form of impredicativity since a function can close over free variables that belong to a higher universe than itself. None of the existing forms of impredicativity (other than those known to be inconsistent) satisfy the needs of closure conversion with a tower of universes. And the last difficulty is that closure conversion exposes internal details of functions, and those details affect the definitional equality of (converted) functions, thereby breaking type preservation.

In this paper we investigate what is necessary to solve those three problems when implementing a type preserving closure conversion for a dependently typed λ-calculus with a tower of universes *without* relying on custom-made constructs in the target language, or more precisely while relying on constructs that are as generic as possible. Concretely, we use equality proofs to solve the first problem, a new form of impredicativity for the second, and quotient types for the third. While these functionalities were not custom-made for the purpose of closure conversion, we show how those high-level features need to be adjusted to accommodate a lower-level language where functions need to be closed.

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 important to ensure that the properties guaranteed by the type checker apply not only to the source code but also to the corresponding compiled code. This gets particularly important for dependently typed programming languages where the programmers invest a lot of effort into embedded proofs in the source code. Currently compilers for dependently typed programs end up throwing away at least some of the type information along the way because we do not know how to preserve it across all the compilation stages. As a consequence, there is no way to check that two separately compiled pieces of code can be linked without breaking any of their invariants, except in indirect ways, such as when they were compiled with the same compiler and we still have access to their source code to check their respective source types.

Preserving type information 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

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

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' types, and hence their own type, which means that the closure converted code requires some form of impredicativity even if the source code was fully predicative. 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 those three hurdles, in order 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 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 extra features would a language need in order to be able to accommodate closure converted code without resorting to a custom construct?

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 this ideal, and gives a possible answer to the previous question.

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 impredicative typing rule for quantification over universe levels, 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.

This work suffers from a significant theoretical weakness in the form of an open question about the soundness of our target language and it may also prove impractical because of a somewhat burdensome encoding and a representation of closures that is not as efficient as that used by real compilers, but we believe that exposing those weaknesses is also a contribution of this work: Why is it that even our best type systems are not able to validate the kind of code our compilers have been generating routinely for decades? Why can't we describe the workings of fully predicative functions without resorting to a brand new notion of impredicativity? Do (non-closed) functions make a type system stronger, in a proof-theoretical sense?

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) [Barendregt 1991] with a tower of universes as our input language:

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

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.

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

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

$\langle \Gamma \rangle$ 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 $\langle x_1, \ldots, x_n \rangle = e$ in e' to mean $e'[e.1/x_1, \ldots, e.n/x_n]$ 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 $x_1, \ldots, x_n$.

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

$$\begin{array}{lll} [\![\lambda x.x + y + z]\!] & = & \langle\ \langle y, z \rangle, \\ & & \quad \lambda\langle x_e, x \rangle.\text{let } \langle y, z \rangle = x_e \text{ in } x + y + z\ \rangle \end{array}$$

This is a pair whose first element holds the "environment" $\langle y, z \rangle$ holding the values of the variables captured by the closure (y and z), and whose second element holds a closed function (the "code"). That function 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:

$$\begin{array}{ll} \langle\ \textit{env} & : \langle y : \mathsf{Int}, z : \mathsf{Int} \rangle, \\ \quad \textit{code} & : \langle x_e : \langle y : \mathsf{Int}, z : \mathsf{Int} \rangle, x : \mathsf{Int} \rangle \to \mathsf{Int}\ \rangle \end{array}$$

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 could use them for that, but we don't want that existentially quantified type to pollute our runtime values. So let's add the following syntax to our terms:

$$\textit{(terms)}\ \ e, \tau\ ::= \ \ldots \mid \exists x : \tau_1.\tau_2 \mid <e_1; e_2> \mid \text{let } <x_1; x_2> = e_1 \text{ in } e_2$$

Here $<e_1; e_2>$ constructs an existential package, and the open eliminator will be constrained to make sure that we can always erase the first element of an existential package. With this new construct, we can now hide the type of the variables captured by our closure:

$$\begin{array}{rcl} [\![\lambda x.x + y + z]\!] & = & < \langle y : \mathsf{Int}, z : \mathsf{Int} \rangle; \\ & & \quad \langle\ \langle y, z \rangle, \\ & & \quad\ \lambda \langle x_e, x \rangle.\mathsf{let}\ \langle y, z \rangle = x_e\ \mathsf{in}\ x + y + z\ \rangle > \\ & : & \exists t : \mathcal{U}.\langle\ env\ :\ t, \\ & & \quad\ code :\ \langle x_e : t, x : \mathsf{Int} \rangle \to \mathsf{Int}\ \rangle \end{array}$$

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 fails to 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 function is equivalent to $\lambda x.x + 7 + z$ but their respective closures will not be equivalent since they don't even have the same size: one captures two variables whereas the other captures only one. Since terms, like those functions, 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 appear in its type, then the simple existential encoding fails. For example, say we have the following primitive:

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

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

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

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

$$\begin{array}{rcl} [\![\lambda x.\mathsf{makevec}\ \alpha\ n\ x]\!] & = & < \langle \alpha : \mathcal{U}, n : \mathsf{Nat} \rangle; \\ & & \quad \langle\ \langle \alpha, n \rangle, \\ & & \quad\ \lambda \langle x_e, x \rangle.\mathsf{let}\ \langle \alpha', n' \rangle = x_e\ \mathsf{in}\ \mathsf{makevec}\ \alpha'\ n'\ x\ \rangle > \\ & : & \exists t : \mathcal{U}.\langle\ env\ :\ t, \\ & & \quad\ code :\ \langle x_e : t, x : \alpha \rangle \to \mathsf{Vec}\ \alpha\ n\ \rangle \end{array}$$

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. With luck, 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, the type needs 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 called *translucent types* [Harper and Lillibridge 1994], and we could also solve it using some form of singleton types, but the more natural solution here is to use equality proofs, which are ubiquitous in dependently typed languages:

$$\textit{(terms)}\ \ e, \tau\ ::=\ \ \ldots\ \mid\ e_1 = e_2\ \mid\ \mathsf{refl}\ \mid\ \mathsf{cast}[e_m]\ e_=\ e$$

$e_1 = e_2$ is the type of equality proofs that e_1 is equal to e_2, refl is the constructor of proofs that $e = e$, and cast is the eliminator which converts e from type $e_m\ e_1$ to type $e_m\ e_2$ when $e_=$ is a proof that $e_1 = e_2$, where e_m is called the *motive* of the elimination. With that, we can fix our conversion:

$$\begin{array}{rcl}
[\![\lambda x.\mathsf{makevec}\ \alpha\ n\ x]\!] & = & < \langle \alpha : \mathcal{U}, n : \mathsf{Nat} \rangle; \\
& & \quad \langle\ \langle \alpha, n \rangle, \\
& & \quad\ \lambda \langle x_e, x, p \rangle.\mathsf{let}\ \langle \alpha', n' \rangle = x_e \\
& & \qquad\qquad \mathsf{in\ let}\ x' = \mathit{cast}[\lambda x'_e.x'_e.1]\ (\mathit{eq_comm}\ p)\ x \\
& & \qquad\qquad\ \mathsf{in\ let}\ \mathit{res} = \mathsf{makevec}\ \alpha'\ n'\ x' \\
& & \qquad\qquad\quad \mathsf{in}\ \mathit{cast}[\lambda x'_e.\mathsf{Vec}\ (x'_e.1)\ (x'_e.2)]\ p\ \mathit{res}\ \rangle > \\
: & \exists t : \mathcal{U}. \langle\ \mathit{env}\ :\ t, & \\
& \quad \mathit{code}\ :\ \langle x_e : t, x : \alpha, p : (x_e = \mathit{env}) \rangle \rightarrow \mathsf{Vec}\ \alpha\ n\ \rangle &
\end{array}$$

On the second line of the type we see that the *code* now takes an additional argument p holding a proof that $x_e = \mathit{env}$. Some readers at this point may be tempted to optimize away x_e since we know it's the same as *env* and refer directly to *env* instead, but that would defeat the purpose since it would result in a non-closed λ-expression. The proof object 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. In the code, we see that p is passed a first time to *cast* (via *eq_comm*, which swaps the terms of the equality) 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 $\alpha'\ n'$) "back" to Vec $\alpha\ n$.

There is one wrinkle remaining here: this approach still would not quite work when the function to be converted is dependently typed. In the example above, the function $\lambda x'_e.\mathsf{Vec}\ (x'_e.1)\ (x'_e.2)$ we pass as the motive of the second *cast* does not need to refer to the argument x, so all is well, but in the general case, the return type may refer to the argument x. But this function we use as motive cannot 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 but x' on the other side. The usual solution to this problem is to merge both *cast*s into one as follows:

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

This is a variant of the *convoy pattern* [Chlipala 2013]: in order to change the type of an expression at the same time as part of its context, we wrap this expression into a function taking the relevant part of its context as an argument (we named this function f' above), and once that function's type is changed (giving us the function f), we pass it the corresponding part of the context as argument.

But we cannot use this solution as-is because f' is not a closed function. The convoy pattern often relies crucially on non-closed functions, but we want to support it in our target language. For this reason, our target language replaces the *cast* operation with a letcast construct which includes the core element of the convoy pattern. It takes the following form:

$$\mathsf{letcast}[e_m, e_=]\ x\ =\ e_1\ \mathsf{in}\ e_2$$

It is equivalent to $(\mathit{cast}[e_m]\ e_=\ (\lambda x.e_2))\ e_1$ except that it avoids the temporary construction of a λ-expression, and thus turns into a plain let after type erasure.

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:

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

Where the ℓ subscript in $\mathcal{U}_\ell$ is the universe level inhabited by the captured environment. This existential type inhabits the universe $\mathcal{U}_{(S\ \ell)}$ since it contains a type $t : \mathcal{U}_\ell$. 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 should thus also be the type above. So we would need to fit into the t field of the existential above an existential 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 might be possible to adapt it to a language with a tower of universes. Yet, we would prefer a solution that does not rely on such custom constructs. Going back to the tuple type above, we can 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 leaks details of the captured variables. As in Sec. 2, 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:

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

This new existential construct $\exists l.\tau$ is kept separate from the previous $\exists x : \tau_1.\tau_2$ because manipulating universe levels as first class values is fraught with danger.

The typing rules will 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. The introduction of universe level variables l is accompanied with a new construct $\ell_1 \sqcup \ell_2$ to get the maximum of two 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^*.

Taking a step back, rather than try and see in which universe we could place a type of the form $\exists l.\tau$, we decided to look at the level of the universe where we need to place our uses of $\exists l.\tau$ in order for our closure conversion to preserve types:

- Let us take a source function type $(x : \tau_1) \to \tau_2\ :\ \mathcal{U}_{\ell_1 \sqcup \ell_2}$, where $\tau_1 : \mathcal{U}_{\ell_1}$ and $\tau_2 : \mathcal{U}_{\ell_2}$.
- In order for our closure conversion to preserve types, we need $[\![(x : \tau_1) \to \tau_2]\!]\ :\ [\![\mathcal{U}_{\ell_1 \sqcup \ell_2}]\!]$. Similarly, we need $[\![\mathcal{U}_\ell]\!] = \mathcal{U}_{[\![\ell]\!]}$ and $[\![S\ \ell]\!] = S\ [\![\ell]\!]$ and $[\![\ell_1 \sqcup \ell_2]\!] = [\![\ell_1]\!] \sqcup [\![\ell_2]\!]$.
- $[\![(x : \tau_1) \to \tau_2]\!]$ is $\exists l.\exists t : \mathcal{U}_l.\langle env : t, code : \langle x_e : t, x : [\![\tau_1]\!], p : (x_e = env)\rangle \to [\![\tau_2]\!]\rangle$.
- Since $[\![\tau_1]\!]\ :\ \mathcal{U}_{[\![\ell_1]\!]}$ and $[\![\tau_2]\!]\ :\ \mathcal{U}_{[\![\ell_2]\!]}$ and the rest fits into $\mathcal{U}_{S\ l}$, the inner $\exists$ has type $\mathcal{U}_{([\![\ell_1]\!] \sqcup [\![\ell_2]\!] \sqcup S\ l)}$.
- So our closure conversion requires that given a type $\tau : \mathcal{U}_{([\![\ell_1]\!] \sqcup [\![\ell_2]\!] \sqcup S\ l)}$, we must have $\exists l.\tau\ :\ \mathcal{U}_{[\![\ell_1]\!] \sqcup [\![\ell_2]\!]}$.

We adopt the rule that $\exists l.\tau$ is given type $\mathcal{U}_{\ell[0/l]}$ when τ has type $\mathcal{U}_\ell$. Whether this choice is sound is currently unknown: impredicativity is notoriously dangerous and this particular form of impredicativity has not been investigated to any significant extent. We discuss in more details the impact of this rule in Sec. 6.5.

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, and since types can contain arbitrary terms, we need to preserve equality between terms, i.e. if $e_1 \simeq e_2$ then $[\![e_1]\!] \simeq [\![e_2]\!]$.

But this is not the case with our current encoding: 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:

$$[\![\lambda x.x + 7]\!]\\ = <1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.x + 7\rangle>>$$

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

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

These two closures are clearly different and it is 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.

With an appropriate model, we could try to leverage parametricity to justify an ad-hoc η-equivalence theorem for our closure objects, as done in [Bowman 2018], but instead we decided to use quotient types, which do not depend so heavily on meta-theoretical properties of our language.

Although support for quotient types in programming languages goes back at least to the first version of Miranda [Thompson 1990; Turner 1985], they are not nearly as common as existentials, tuples, and equality proofs. There are different ways to add quotient types to a language, but to a first approximation we can classify them into two groups: those which define a quotient via a relation (and hence correspond most closely to the traditional mathematical presentation) such as higher inductive types [The Univalent Foundations Program 2013], and those which define a quotient via a normalization function that takes elements to their equivalence class [Courtieu 2001].[1] While the use of relations is arguably more standard and general, we opted to use quotient types based on normalization functions, because they provide a stronger definitional equality, which significantly simplifies our proof of type preservation because definitional equality is thus also preserved by the conversion. More specifically, we add the following terms to our syntax:

$$\textit{(terms)}\ \ e, \tau\ ::=\ \ \ldots\ \mid\ \mathsf{Q}\ e_f\ \mid\ \mathsf{Qin}[e_f]\ e\ \mid\ \mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2$$

Where $\mathsf{Q}\ e_f$ is the quotient type defined by the normalization function e_f of type $\tau \to \tau$, where τ is the type that is quotiented by e_f; $\mathsf{Qin}[e_f]\ e$ is the constructor which projects values of type τ into $\mathsf{Q}\ e_f$; and $\mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2$ is the elimination form where $e_=$ is the proof that we obey the quotient's equality, i.e. a proof that $e_2 \simeq e_2[e_f\ x/x]$. There is no dedicated introduction form for equality between quotiented values, because instead we strengthen the definitional equality so $\mathsf{Qin}[e_f]\ e_1 \simeq \mathsf{Qin}[e_f]\ e_2$ when $e_f\ e_1 \simeq e_f\ e_2$.

So we can now quotient the encoding of our closures with a normalization function. We choose as canonical representative or each equivalence class the closure object where the captured environment is empty. These representative closure objects have the property that their *code* is in general not closed any more, and indeed they amount, in a sense, to turning the closures back into their non-closed form, thereby recovering the original equality semantics:

$$\begin{array}{l}[\![\tau_1 \to \tau_2]\!]_{\text{inner}} = \exists l.\exists t : \mathcal{U}_l.\langle\ env\ :\ t, \\ \qquad\qquad\qquad\qquad\qquad\qquad code :\ \langle x_e : t, x : [\![\tau_1]\!], p : (x_e = env)\rangle \to [\![\tau_2]\!]\ \rangle) \\ [\![\tau_1 \to \tau_2]\!] \\ = \mathsf{Q}\ (\lambda<l; <t; \langle env, code\rangle>>.<1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.code\ \langle env, x, \mathsf{refl}\rangle\rangle>> \\ \qquad : [\![\tau_1 \to \tau_2]\!]_{\text{inner}} \to [\![\tau_1 \to \tau_2]\!]_{\text{inner}})\end{array}$$

When we want to call a closure, we need to use the elimination form of the quotient $\mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2$ to which we have to provide the proof $e_=$ that we obey the quotient's equality. For our uses,

[1] With various options trying to provide a mix of the two, see for instance the quotient library by Cohen [2013], or the Arend proof assistant.

this proof is simple since e_2 is the code that calls the closure and it is very similar to the code of the normalization function.

When defining quotient types using axioms, the quotient eliminator is typically defined to take the quotiented object and a function representing the body of our let construct, but of course, this would not work here because that function would usually not be closed. In other words, our eliminator takes the shape of a let for the same reason we introduced letcast. This is a common theme when working in a lower-level intermediate language.

3.4 Taming closedness

One more issue that keeps popping up is what exactly we should consider as "closed": given a source function $\lambda x : \tau . x\ y$, it is natural to consider that we should make a closure that captures y and nothing else, but after closure conversion the code will usually want to refer to the types of both x and y, for example in the motives of the casts.

Also, dependencies can get in the way. Let's say we have a source function $\lambda x . insert\ t\ x\ s$ with a typing environment that contains $\{..., t : \mathcal{U}_1, o : Ordering\ t, s : BinarySet\ t\ o, ...\}$. If we close over t and s but not over o, the closed code will not be properly typed because the type of s will be ill-formed since it will not refer to the same t as the type of o any more. For this reason, we need to transitively close over all the variables that appear in the types of the free variables (as well as the type of the function itself).

More problematic yet: in the previous section, the quotient's normalization function includes the non-closed function $\lambda \langle x_e, x, p \rangle . code\ \langle env, x, \mathsf{refl} \rangle$. We cannot easily close this function without reintroducing the problem it's trying to address, nor can we hide it via some kind of let-like construct.

For those reasons, we need to refine what we mean by closed: we need to distinguish those elements needed only for typing purposes from those needed at run time: our closure conversion does not require that all λ-expressions be closed in the closure-converted code, but only that all λ-expressions be closed after type erasure.

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:

$$
\begin{array}{llll}
\textit{(levels)} & \ell & ::= & S\ 0 \mid S\ \ell \\
\textit{(sorts)} & s & ::= & \mathcal{U}_\ell \\
\textit{(neutral terms)} & e, \tau, N & ::= & x \mid s \mid (M : \tau) \mid (x : \tau_1) \to \tau_2 \mid N\ M \\
\textit{(normal terms)} & e, M & ::= & N \mid \lambda x . M
\end{array}
$$

This is the same language as shown at the beginning of Section 2, except that the bottom universe level 1 is now spelled $S\ 0$ and terms are split into neutral terms, for which types can be synthesized, and normal terms, for which types have to be checked. We still use e in most places except when this distinction is important, and we use τ for terms that are intended to denote types. The use of $S\ 0$ for the bottom universe is there solely so as to keep the universe levels of the source language aligned with those of the target language where we will need an additional "basement" universe 0. The typing rules for this source language are shown in Figure 1. We show them in the bidirectional style [Dunfield and Krishnaswami 2021], so as to try and reduce the type annotations in our code. So the main judgment is split between $\Gamma \vdash e \Rightarrow \tau$ and $\Gamma \vdash e \Leftarrow \tau$ where the first synthesizes the

$\boxed{\Gamma \vdash N \;\Rightarrow\; \tau}$ and $\boxed{\Gamma \vdash M \;\Leftarrow\; \tau}$ The form has type τ in context Γ:

$$\Gamma \vdash \mathcal{U}_\ell \;\Rightarrow\; \mathcal{U}_{(S\,\ell)} \qquad \frac{\tau_1 \rightsquigarrow^* \tau_2 \qquad \Gamma \vdash e \;\Leftarrow\; \tau_2}{\Gamma \vdash e \;\Leftarrow\; \tau_1}\;(\textsc{Red}_C) \qquad \frac{\Gamma \vdash e \;\Rightarrow\; \tau_1 \qquad \tau_1 \rightsquigarrow^* \tau_2}{\Gamma \vdash e \;\Rightarrow\; \tau_2}\;(\textsc{Red}_S)$$

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

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \;\Rightarrow\; \tau}\;(\textsc{Var}) \qquad \frac{\Gamma \vdash \tau_1 \;\Rightarrow\; \mathcal{U}_{\ell_1} \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 \;\Rightarrow\; \mathcal{U}_{\ell_2} \qquad \ell_3 = \max(\ell_1, \ell_2)}{\Gamma \vdash (x{:}\tau_1) \to \tau_2 \;\Rightarrow\; \mathcal{U}_{\ell_3}}\;(\textsc{Pi})$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \;\Leftarrow\; \tau_2}{\Gamma \vdash \lambda x.e \;\Leftarrow\; ((x{:}\tau_1) \to \tau_2)}\;(\textsc{Lam}) \qquad \frac{\Gamma \vdash e_1 \;\Rightarrow\; (x{:}\tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 \;\Leftarrow\; \tau_1}{\Gamma \vdash e_1\; e_2 \;\Rightarrow\; \tau_2[(e_2 : \tau_1)/x]}\;(\textsc{App})$$

Fig. 1. Typing rules of the source language.

$\boxed{e_1 \simeq e_2}$ e_1 is definitionally equal to e_2:
$\boxed{e_1 \rightsquigarrow e_2}$ e_1 reduces to e_2:

$$(N : \tau) \;\rightsquigarrow\; N \qquad ((\lambda x.e_1) : (x{:}\tau_1) \to \tau_2)\; e_2 \;\rightsquigarrow\; (e_1 : \tau_2)[(e_2 : \tau_1)/x]$$

$$\frac{e \;\rightsquigarrow\; e'}{E[e] \;\rightsquigarrow\; E[e']} \qquad \frac{e_1 \;\rightsquigarrow^*\; e_3 \qquad e_2 \;\rightsquigarrow^*\; e_3}{e_1 \simeq e_2}$$

$$\textit{(eval contexts)} \quad E \;::=\; \bullet \mid (E : \tau) \mid (e : E) \mid (x{:}E) \to \tau \mid (x{:}\tau) \to E \mid E\; e \mid e\; E \mid \lambda x.E$$

Fig. 2. Convertibility rules of our source language.

type τ from the context Γ and the term e whereas the second just checks it. The rules presume that Γ in the conclusion is well-formed, as is the type passed to the $\Leftarrow$ rule. The two Red rules allow arbitrary reductions between typing steps, but the intention is to use them only when needed to reduce to weak-head normal form. Our calculus does not use universe subsumption, which is why in the Pi rule, we allow input and output types from different universes.

The convertibility rules $e_1 \simeq e_2$ and $e_1 \rightsquigarrow e_2$ are shown in Figure 2. The rules are untyped but defined such that if input expressions are properly typed in the same context, then all the terms used along the way are themselves properly typed. To that end, $e_1 \simeq e_2$ is defined on top of $\rightsquigarrow^*$ which is the transitive reflexive closure of small step reduction rule $\rightsquigarrow$. The only typing rule which uses the convertibility rule, i.e. the Conv rule, indeed guarantees that τ_1 and τ_2 are well formed in the same context, so in practice $e_1 \simeq e_2$ will always be invoked with well-formed terms. Similarly $e_1 \rightsquigarrow e_2$ is only ever used with a well typed e_1.

Beside the details of presentation of conversion and type checking, this is a conventional dependently typed λ-calculus with a predicative tower of universes and without universe subsumption. It enjoys a lot of nice meta-theoretical properties, but the only one that we will really need here is subject reduction.

(levels)	ℓ	::=	$l \mid 0 \mid S\ \ell \mid \ell_1 \sqcup \ell_2$	
(sorts)	s	::=	$\mathcal{U}_\ell$	
(index)	i	$\in$	$\mathbb{N}^*$	Positions in tuples
(ctx)	Γ	::=	$\bullet \mid \Gamma, x : \tau$	
(neutral terms)	e, τ, N	::=	$x \mid s$	Variables and sorts
			$\mid (M : \tau)$	Type annotation
			$\mid (x : \tau_1) \to \tau_2$	Function type
			$\mid N\ M$	Function application
			$\mid \langle \Gamma \rangle$	Dependent tuple type
			$\mid N.i$	i^{th} projection from a tuple
			$\mid \exists x : \tau_1 . \tau_2$	Existential type
			$\mid$ let $<x_1; x_2> = N_1$ in N_2	Existential eliminator
			$\mid \exists l . \tau$	Existential universe type
			$\mid$ let $<l; x> = N_1$ in N_2	Existential universe eliminator
			$\mid$ Eq $N\ M$	Equality type
			$\mid$ letcast$[N_m, N_=]\ x\ =\ N$ in M	Eliminator of equality type
			$\mid$ Q N	Quotient type
			$\mid$ let$[M_=]$ Qin $x = N_1$ in N_2	Quotient eliminator
			$\mid$ Qin$[N_n]\ N$	Constructor of quotient
			$\mid$ Qn$[N_n]\ N$	Auxiliary constructor of quotient
(normal terms)	e, M	::=	N	Neutral term
			$\mid \lambda x . M$	Function constructor
			$\mid \langle M_1, ..., M_n \rangle$	Tuple constructor
			$\mid <M_1; M_2>$	Existential constructor
			$\mid <\ell; M>$	Existential universe constructor
			$\mid$ refl	Equality constructor

Fig. 3. Syntax of the target language

Lemma 4.1 (Subject reduction).
Given a well-formed context Γ*, if* $\Gamma \vdash e_1 \Leftarrow \tau$ *and* $e_1 \rightsquigarrow e_2$ *then* $\Gamma \vdash e_2 \Leftarrow \tau$.

Proof. By induction on the derivation of $e_1 \rightsquigarrow e_2$, using the usual substitution lemma. □

4.2 Target language syntax

Our target language is a superset of our source language. Its syntax is given in Figure 3. The quantification over universe levels adds two new elements to the syntax of levels ℓ, one for level variables l, and another for the maximum of two levels $\ell_1 \sqcup \ell_2$. The sorts and contexts stay unchanged, but we add many new elements to the terms:

- Dependent tuples: $\langle \Gamma \rangle$ 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.
- Existential types: $\exists x : \tau_1 . \tau_2$ is the existential type that quantifies over x; $<e_1; e_2>$ is the corresponding constructor; and let $<x_1; x_2> = e_1$ in e_2 is its elimination form, where x_1 can be used only in erasable positions, such as type annotations.
- Existential universe types: $\exists l . \tau$ is the existential type that quantifies over universe level l; $<\ell; e>$ is the corresponding constructor; and let $<l; x> = e_1$ in e_2 is its elimination form.

$$
\begin{array}{lcl}
\mathsf{cast}[x_m] & : & \mathsf{Eq}\ e_1\ e_2 \to x_m\ e_1 \to x_m\ e_2 \\
\mathsf{cast}[x_m]\ x_=\ e & = & \mathsf{letcast}[\lambda x.\langle\bullet\rangle \to x_m\ x, x_=]\ \langle\rangle\ =\ \langle\rangle\ \mathsf{in}\ e \\
\mathsf{call} & : & [\![(x:\tau_1) \to \tau_2]\!] \to (x:[\![\tau_1]\!]) \to [\![\tau_2]\!] \\
\mathsf{call}\ c\ x & = & \mathsf{let}\ <l; <t; \langle env, code\rangle >> = c\ \mathsf{in}\ code\ \langle env, x, \mathsf{refl}\rangle \\
\mathsf{tfv}(\Gamma, e) & = & \bigcup_{x\in\mathsf{fv}(e)}\{x\} \cup \mathsf{tfv}(\Gamma, \Gamma(x)) \\
[\![(x:\tau_1) \to \tau_2]\!]_{\mathrm{inner}} & = & \exists l.\exists t : \mathcal{U}_l.\langle\ env\ :\ t, \\
 & & \qquad code :\ \langle x_e : t, x : [\![\tau_1]\!], p : (x_e = env)\rangle \to [\![\tau_2]\!]\ \rangle \\
[\![(x:\tau_1) \to \tau_2]\!]_{\mathrm{norm}} & = & (\lambda c.<1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x\rangle >> \\
 & & \quad : [\![(x:\tau_1) \to \tau_2]\!]_{\mathrm{inner}} \to [\![(x:\tau_1) \to \tau_2]\!]_{\mathrm{inner}})
\end{array}
$$

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![\mathcal{U}_\ell]\!] & = & \mathcal{U}_\ell \\
[\![(e:\tau)]\!] & = & ([\![e]\!] : [\![\tau]\!]) \\
[\![(x:\tau_1) \to \tau_2]\!] & = & \mathsf{Q}\ [\![(x:\tau_1) \to \tau_2]\!]_{\mathrm{norm}} \\
[\![e_1\ e_2]\!] & = & \mathsf{let}[\mathsf{refl}]\ \mathsf{Qin}\ c = [\![e_1]\!]\ \mathsf{in}\ \mathsf{call}\ c\ [\![e_2]\!] \\
[\![\lambda x_a.e]\!] & = & \mathsf{Qin}[[\![(x_a:\tau_a) \to \tau_r]\!]_{\mathrm{norm}}] \\
 & & \quad (<\ell; <\langle[\![\Gamma']\!]\rangle; \langle\langle\vec{x}\rangle, \lambda\langle x_e, x'_a, x_=\rangle.body\rangle >> : [\![(x_a:\tau_a) \to \tau_r]\!]_{\mathrm{inner}}) \\
 & & \mathsf{where}\ \ \Gamma \vdash \lambda x_a.e \Leftarrow (x_a:\tau_a) \to \tau_r \qquad \text{Extract the type information} \\
 & & \quad \vec{x_f}\ =\ \mathsf{tfv}(\Gamma, \lambda x_a.e) \cup \qquad \text{Find the free variables} \\
 & & \qquad\quad \mathsf{tfv}(\Gamma, (x_a:\tau_a) \to \tau_r) \\
 & & \quad \Gamma'\ =\ \Gamma|_{\vec{x_f}}\ =\ \vec{x} : \vec{\tau} \qquad \text{Get their type and order them} \\
 & & \quad \Gamma \vdash \langle\Gamma'\rangle\ \Rightarrow\ \mathcal{U}_\ell \qquad \text{Compute the universe level } \ell \\
 & & \quad f_m\ =\ \lambda\langle\vec{x}\rangle.(x_a:[\![\tau_a]\!]) \to [\![\tau_r]\!] \qquad \text{The } \textit{motive} \text{ of the cast} \\
 & & \quad body = \mathsf{letcast}[f_m, x_=]\ x_a\ =\ x'_a\ \mathsf{in} \\
 & & \qquad\quad \mathsf{let}\ \langle\vec{x}\rangle = x_e\ \mathsf{in}\ [\![e]\!]
\end{array}
$$

Fig. 4. The closure conversion itself

- Equality types: $\mathsf{Eq}\ e_1\ e_2$ is the type of proofs that e_1 and e_2 are equal; refl is the corresponding constructor of the proof by reflexivity; and $\mathsf{letcast}[e_m, e_=]\ x\ =\ e_1\ \mathsf{in}\ e_2$ is the elimination form which takes a proof $e_= : \mathsf{Eq}\ e_i\ e_o$, a motive e_m, and behaves like $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ except that e_2 is typed in a context where some e_o are replaced by e_i, according to e_m.
- Quotient types: $\mathsf{Q}\ e$ is the quotient type where e is the normalization function; $\mathsf{Qin}[e_n]\ e$ is its main constructor where e_n is again the normalization function; and $\mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2$ is its elimination form where $e_=$ is the proof that it obeys the quotient's equality. We follow the design of *normalizable types* [Monnier 2024] and thus need a secondary constructor $\mathsf{Qn}[e_n]\ e$ used only internally in the typing and reduction rules to keep track of the fact that e has already been normalized.

As is customary for bidirectional typing, all the types and eliminators are neutral terms, and most of the constructors are normal terms. Qin is nevertheless an exception to this rule: we could make it a normal term of the form $\mathsf{Qin}\ e$ and determine e_n from its expected return type, but we need e_n in the reduction rules of Qin, so we opted to include e_n in the syntax $\mathsf{Qin}[e_n]\ e$ in order to avoid the need for a typed conversion rule. The same argument does not apply to Qn, but we kept it as a neutral term simply to minimize the difference with Qin.

4.3 Closure conversion

We can now show the actual closure conversion itself, denoted $[\![\cdot]\!]$, which is in Figure 4. In that figure, we use 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.

We use the following auxiliary definitions:

- $\mathsf{cast}[x_m]\ x_=\ e$: The "usual" equality type elimination, defined on top of letcast.
- $\mathsf{call}\ c\ x$: The code of a call to the non-quotiented closure c with argument x.
- $\mathsf{tfv}(\Gamma, e)$: The set of transitively free variables, which includes the free variables of the types of the free variables.
- $[\![(x:\tau_1) \to \tau_2]\!]_{\text{inner}}$: The *non-quotiented* type of a closure, for a source type $(x:\tau_1) \to \tau_2$.
- $[\![(x:\tau_1) \to \tau_2]\!]_{\text{norm}}$: The normalization function for the quotient type used on functions of source type $(x:\tau_1) \to \tau_2$. This returns the function fully annotated with its type since it will be used in places where we need to be able to synthesize the type.

The first three cases of the closure conversion itself 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 quotiented 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 first looks inside the quotiented object and then uses call to perform the actual function invocation which proceeds by unpacking the 4-tuple to then invoke the *code* with the *env*, the actual argument, and the trivial refl proof that the first argument is indeed *env*. The proof that this obeys the quotient's equality is provided by another refl because $\mathsf{call}\ ([\![\ldots]\!]_{\text{norm}}\ c)\ e$ simply reduces to $\mathsf{call}\ c\ e$: all the constructors introduced by $[\![\ldots]\!]_{\text{norm}}$ find their corresponding elimination forms in call.

The hard work is all concentrated in the case for $\lambda x_a.e$: there we actually build the closure object, made of its quotiented existential nested tuples, as well as the closed code:

- x_e is the formal variable that will hold the tuple of captured variables.
- x_a is the function's formal argument. In the source code its type is defined in the ambient context Γ of the function, obviously. But in the converted code, this is not so simple because from the outside of the closure we also want the type to refer to elements of the ambient context Γ, but the body of the code cannot and has to refer to those same elements via the argument x_e instead, which will contain the captured environment. For this reason, in the converted code, the source x_a ends up split into x'_a with the "outside" type and x_a with the "inside" type.
- $\vec{x}_f$ is the set of captured variables, in no particular order. We compute it using tfv which gives us the set of transitively free variables.
- $\vec{x}$ is this same set but properly ordered according to the order in which they appear in the environment, so that dependencies between them are obeyed.
- $\langle [\![\Gamma']\!] \rangle$, also known as $\langle \vec{x} : [\![\vec{\tau}]\!] \rangle$, is the type of the *env* tuple holding the captured variables.
- ℓ is the universe level of $\langle [\![\Gamma']\!] \rangle$.
- The closed function then just takes its three arguments x_e, x'_a, and $x_=$ (seen on the 2nd line of the code), after which *body* unpacks the environment x_e with a let which (re)reintroduces all the free variables expected by the original body of the function and then evaluates the

$$\textit{(level contexts)} \quad L ::= \bullet \mid S\ L \mid L \sqcup \ell \mid \ell \sqcup L$$
$$\textit{(eval contexts)} \quad E ::= \ldots \mid \mathcal{U}_L$$

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

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

$\boxed{|e|}$ Type erasure of e:

$$\begin{array}{lcl} |x| & = & x \\ |\mathcal{U}_\ell| & = & \mathcal{U} \\ |(e : \tau)| & = & |e| \\ |(x{:}\tau_1) \to \tau_2| & = & \to \\ |e_1\ e_2| & = & |e_1|\ |e_2| \\ |\lambda x.e| & = & \lambda x.|e| \end{array}$$

Fig. 5. Differences with source language for the core target language.

closure converted body $[\![e]\!]$, while carefully using $x_=$ to mediate between the "inside" types and the "outside" types.

4.4 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 5 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. Another change is the fact that the base universe level is now really called 0, so it sits one level below the base universe of the source language.

In addition to those changes, the target language has a notion of type erasure denoted $|e|$ whose definition for the base elements of the language is shown in that same figure, where we can see that it simply traverses the terms and removes the explicit type annotations and the universe levels. We will see soon that it has further effects on other constructs.

4.4.1 Dependent tuples. Figure 6 shows the rules governing the dependent tuples. There is nothing novel here. The main complexity lies in the rules Proj and Tup which need to take into account the possible dependencies between the fields, which requires applying substitutions to replace references to previous fields with those fields' values when returning the type of fields. Notice also the Σ rule which computes the maximum level of universe in the tuples' members in order to decide in which universe to put the tuple type.

4.4.2 Existential types. Figure 7 shows the rules that govern existential types. The first thing to note here is in the top right corner, we see that type erasure throws away the left hand side of those existential packages. For this to make sense, the Open rule makes sure the x_1 variable is not used in a way that is computationally relevant, by enforcing that it does not occur as a free variable in the *type erasure* of the body of the let. This is the only place where the type erasure is used in the typing rules.

In most other respects, this is otherwise a simplified version of the rules we use for dependent tuples, where the simplification comes from the fact that we only handle pairs here, instead of arbitrary number of elements, and the fact that the let elimination term does not allow the return type to depend on the pair. Regarding this Open rule, the $\Gamma \vdash \tau \Rightarrow s$ premise may seem redundant

$$\begin{array}{llll} \textit{(terms)} & e,\tau & ::= & \ldots \mid \langle\Gamma\rangle \mid \langle e_1,..,e_n\rangle \mid e.i \\ \textit{(eval ctxts)} & E & ::= & \ldots \mid \langle\Gamma_1, x:E, \Gamma_2\rangle \mid \langle \vec{e_b}, E, \vec{e_a}\rangle \mid E.i \end{array} \qquad \begin{array}{lcl} |\langle\Gamma\rangle| & = & \langle\bullet\rangle \\ |\langle e_1,..,e_n\rangle| & = & \langle |e_1|,..,|e_n|\rangle \\ |e.i| & = & |e|.i \end{array}$$

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

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

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

$$(\langle\vec{e}\rangle : \langle\vec{x}:\vec{\tau}\rangle).i \rightsquigarrow (e_i : \tau_i[(e_1:\tau_1)/x_1, ..., (e_{i-1}:\tau_{i-1})/x_{i-1}])$$

Fig. 6. Dependent tuples.

$$\begin{array}{llll} \textit{(terms)} & e,\tau & ::= & \ldots \mid \exists x:\tau_1.\tau_2 \mid <e_1;e_2> \\ & & & \mid \mathsf{let} <x_1;x_2> = e_1 \mathsf{\ in\ } e_2 \\ \textit{(eval ctxts)} & E & ::= & \ldots \mid \exists x:E.\tau \mid \exists x:\tau.E \\ & & & \mid <E;e> \mid <e;E> \\ & & & \mid \mathsf{let} <x_1;x_2> = E \mathsf{\ in\ } e \\ & & & \mid \mathsf{let} <x_1;x_2> = e \mathsf{\ in\ } E \end{array} \qquad \begin{array}{lcl} |\exists x:\tau_1.\tau_2| & = & \exists \\ |<e_1;e_2>| & = & |e_2| \\ |\mathsf{let} <x_1;x_2> = e_1 \mathsf{\ in\ } e_2| & & \\ \quad = \mathsf{let}\ x_2 = |e_1| \mathsf{\ in\ } |e_2| & & \end{array}$$

$$\frac{\Gamma \vdash \tau_1 \Rightarrow \mathcal{U}_{\ell_1} \qquad \Gamma, x:\tau_1 \vdash \tau_2 \Rightarrow \mathcal{U}_{\ell_2}}{\Gamma \vdash \exists x:\tau_1.\tau_2 \Rightarrow \mathcal{U}_{\ell_1 \sqcup \ell_2}}\ (\exists)$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \Gamma \vdash e_2 \Leftarrow \tau_2[(e_1:\tau_1)/x]}{\Gamma \vdash <e_1;e_2> \Leftarrow \exists x:\tau_1.\tau_2}\ (\textsc{Pack})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \exists x_1:\tau_1.\tau_2 \qquad x_1 \notin \mathsf{fv}(|e_2|) \qquad \Gamma, x_1:\tau_1, x_2:\tau_2 \vdash e_2 \Rightarrow \tau \qquad \Gamma \vdash \tau \Rightarrow s}{\Gamma \vdash \mathsf{let} <x_1;x_2> = e_1 \mathsf{\ in\ } e_2 \Rightarrow \tau}\ (\textsc{Open})$$

$$\mathsf{let} <x_1;x_2> = (<e_1;e_2> : \exists x_1:\tau_1.\tau_2) \mathsf{\ in\ } e_b \quad \rightsquigarrow \quad e_b[(e_1:\tau_1)/x_1, (e_2:\tau_2[(e_1:\tau_1)/x_1])/x_2]$$

Fig. 7. Existential types.

since we already know that τ is a well-formed type and we don't use s elsewhere, but its purpose is to verify that τ is closed w.r.t. x_1 and x_2.

These rules can be seen as a subset of those proposed by Bernardo [2009] for the Σ type of ICC^*_Σ. Our $\exists x:\tau_1.\tau_2$ corresponds in that work to $\Sigma[x:\tau_1].\tau_2$, i.e. a pair whose first field is erased, except that we use a weaker rule for the eliminator: the eliminator for ICC^*_Σ is dependent, in the sense that it allows the return type to depend on the pair, whereas we use a simpler non-dependent eliminator.

$$
\begin{array}{lrcl}
\textit{(terms)} & e, \tau & ::= & \ldots \mid \exists l.\tau \mid \langle \ell; e \rangle \\
 & & \mid & \mathsf{let}\ \langle l; x \rangle = e_1\ \mathsf{in}\ e_2 \\
\textit{(eval ctxts)} & E & ::= & \ldots \mid \exists l.E \mid \langle L; e \rangle \mid \langle \ell; E \rangle \\
 & & \mid & \mathsf{let}\ \langle l; x \rangle = E\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ \langle l; x \rangle = e\ \mathsf{in}\ E
\end{array}
$$

$$
\begin{array}{l}
|\exists l.\tau| \ = \ \exists \\
|\langle \ell; e \rangle| \ = \ |e| \\
|\mathsf{let}\ \langle l; x \rangle = e_1\ \mathsf{in}\ e_2| \\
\qquad = \ \mathsf{let}\ x = |e_1|\ \mathsf{in}\ |e_2|
\end{array}
$$

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

$$
\frac{\Gamma \vdash e_1 \ \Rightarrow\ \exists l.\tau_1 \qquad l \notin \mathsf{fv}(\Gamma) \cup \mathsf{fv}(\tau) \qquad \Gamma, x : \tau_1 \vdash e_2 \ \Rightarrow\ \tau \qquad \Gamma \vdash \tau \ \Rightarrow\ s}{\Gamma \vdash \mathsf{let}\ \langle l; x \rangle = e_1\ \mathsf{in}\ e_2 \ \Rightarrow\ \tau}\ (\text{U-Open})
$$

$$
\mathsf{let}\ \langle l; x \rangle = (\langle \ell; e_1 \rangle : \exists l.\tau)\ \mathsf{in}\ e_2 \quad \rightsquigarrow \quad e_2[\ell/l, (e_1 : \tau[\ell/l])/x]
$$

Fig. 8. Existential universe types.

4.4.3 Existential universe types. Figure 8 shows the rules that govern existential quantification over universe levels. We see that the type erasure for existential universe types follow basically the same rules as for existential types. But note that U-Open does not need to check that l is not used in the erasure of e_2 because we know it by construction: erasure erases all universe level annotations anyway.

Our typing judgment does not include any context listing the set of universe level variables l that are currently in scope. There is no deep technical reason for that, we simply decided to go for a presentation that does without it, as is done sometimes for the set of type variables in 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. It does require extra care in the typing rules to avoid variable captures since we cannot rely on the usual Barendregt convention. This manifests itself in the tests that l is not free in Γ or τ in the U-$\exists$ and U-Open rules.

More importantly we see in the U-$\exists$ rule the crucial impredicativity where we return the type $\mathcal{U}_{\ell[0/l]}$, which corresponds to the infimum $\mathsf{inf}_l\ \ell$ whereas the predicative choice would be to use the supremum $\mathsf{sup}_l\ \ell$ which would result in something like $\mathcal{U}_\omega$, as used in Agda. This is the only part of our target language that is fundamentally unique; most of the rest are minor variations of well understood constructs found in other languages.

It makes our language impredicative but in a way that is qualitatively different from the traditional notion of impredicative universes like Prop: on the one hand, the notion impredicativity introduced here affects all universes rather than only specific ones, and on the other hand it is less direct: one cannot directly quantify over a larger universe as we do for quantification in Prop, instead one is restricted to quantify over a universe which is itself quantified over all levels: so while that universe may be larger it is not guaranteed to be larger.

Another detail to note is that in practice $\exists l.\tau$ will almost always belong to a universe above $\mathcal{U}_0$ because, in order to be of any use τ will inevitably contain something like $\exists t : \mathcal{U}_l.\tau'$, so τ will belong to a universe $\mathcal{U}_{S\ l}$ or larger. This applies in particular to our closures. For this reason in our closure conversion, we keep the universe level 0 basically unused. This is also why we use the universe level 1 as our base level in the source language. Of course, we could use 0 as the base level in both languages and map source code from universe level ℓ to target code in universe level $S\ \ell$, i.e. $[\![\mathcal{U}_\ell]\!] = \mathcal{U}_{S\ \ell}$.

$$
\begin{array}{lrcl}
\textit{(terms)} & e, \tau & ::= & \ldots \mid \mathsf{Eq}\ e_1\ e_2 \mid \mathsf{refl} \\
 & & \mid & \mathsf{letcast}[e_m, e_=]\ x\ =\ e_1\ \mathsf{in}\ e_2 \\
\textit{(eval ctxts)} & E & ::= & \ldots \mid \mathsf{Eq}\ E\ e \mid \mathsf{Eq}\ e\ E \\
 & & \mid & \mathsf{letcast}[e_m, e_=]\ x\ =\ E\ \mathsf{in}\ e_2 \\
 & & \mid & \mathsf{letcast}[e_m, e_=]\ x\ =\ e_1\ \mathsf{in}\ E \\
 & & \mid & \mathsf{letcast}[E, e_=]\ x\ =\ e_1\ \mathsf{in}\ e_2 \\
 & & \mid & \mathsf{letcast}[e_m, E]\ x\ =\ e_1\ \mathsf{in}\ e_2
\end{array}
$$

$$
\begin{array}{l}
|\mathsf{Eq}\ e_1\ e_2| \ =\ \mathsf{Eq} \\
|\mathsf{refl}| \quad\ =\ \mathsf{refl} \\
|\mathsf{letcast}[e_m, e_=]\ x\ =\ e_1\ \mathsf{in}\ e_2| \\
\qquad =\ \mathsf{let}\ x = |e_1|\ \mathsf{in}\ |e_2|
\end{array}
$$

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

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

$$
\frac{e_m\ e_1 \simeq (x{:}\tau_a) \to \tau_r}{\mathsf{letcast}[e_m, (\mathsf{refl} : \mathsf{Eq}\ e_1\ e_2)]\ x\ =\ e_a\ \mathsf{in}\ e_r \quad \rightsquigarrow \quad (e_r : \tau_r)[e_a/x]}
$$

Fig. 9. Typing rules for the identity type.

4.4.4 Equality types. Figure 9 shows the rules governing the equality (also called identity) type. The Eq and Refl rules are fairly standard, and while the letcast primitive is non-standard due to the need to accommodate the constraint that we need 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 [Chlipala 2013]. Similarly the reduction rule is made more complex by the needs to compute the type annotation in the result. Since the reduction rules are actually untyped, an actual implementation could strip all type annotations while performing conversion checks.

More importantly, we see in the erasure rules that the letcast is converted to a plain let, since the equality proof $e_=$ and the elimination motive e_m are not needed at run time.

4.4.5 Quotient types. Figure 10 shows the rules for our quotient types, which follow the design of normalizable types [Monnier 2024]. The type erasure rules again turn the elimination into a plain let, and they also simple erase the two constructors, so we see that all the existential packages, all the quotienting and its normalization functions, and all the equality proofs disappear after type erasure as one would expect.

A type $\mathsf{Q}\ e_n$ is the type of values of type τ_1 quotiented by the normalization function e_n which returns the canonical representative of each equivalence class. $\mathsf{Qin}[e_n]\ e_v$ is the normal constructor and behaves a bit like a "smart" constructor in that it comes with a reduction rule which immediately normalizes it to $\mathsf{Qn}[e_n]\ (e_n\ e_v)$. If we consider only the typing rules, such quotients may seem useless: we could just replace $\mathsf{Qin}[e_n]\ e_v$ with $e_n\ e_v$ and do away with all the Q and Qn, and the code would still type-check. The benefit of our quotient types is elsewhere: since the erasure of $\mathsf{Qin}[e_n]\ e_v$ is just $|e_v|$, the normalization does not take place during run-time.

This distinction is crucial for us because our normalization functions return closure objects containing a non-closed function, so using them at run-time would be simply impossible (or at least it would defeat the purpose of the closure conversion).

$$
\begin{array}{lrcl}
\textit{(terms)} & e, \tau & ::= & \ldots \mid \mathsf{Q}\ e_n \mid \mathsf{Qin}[e_n]\ e_v \mid \mathsf{Qn}[e_n]\ e_v \\
& & \mid & \mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2 \\
\textit{(eval ctxts)} & E & ::= & \ldots \mid \mathsf{Q}\ E \mid \mathsf{Qn}[E]\ e \mid \mathsf{Qn}[e]\ E \\
& & \mid & \mathsf{let}[E]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2 \\
& & \mid & \mathsf{let}[e_=]\ \mathsf{Qin}\ x = E\ \mathsf{in}\ e_2 \\
& & \mid & \mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ E
\end{array}
$$

$$
\begin{array}{rcl}
|\mathsf{Q}\ e_n| & = & \mathsf{Q} \\
|\mathsf{Qin}[e_n]\ e_v| & = & |e_v| \\
|\mathsf{Qn}[e_n]\ e_v| & = & |e_v| \\
|\mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2| & & \\
\quad = \ \mathsf{let}\ x = |e_1|\ \mathsf{in}\ |e_2| & &
\end{array}
$$

$$
\frac{\Gamma \vdash e_n \Rightarrow \tau \to \tau \qquad \Gamma \vdash \tau \Rightarrow s}{\Gamma \vdash \mathsf{Q}\ e_n \Rightarrow s}\ (\mathsf{Q}) \qquad \frac{\Gamma \vdash e_n \Rightarrow \tau \to \tau \qquad \Gamma \vdash e_v \Leftarrow \tau}{\Gamma \vdash \mathsf{Qin}[e_n]\ e_v \Rightarrow \mathsf{Q}\ e_n}\ (\textsc{Qin})
$$

$$
\frac{\Gamma \vdash e_n \Rightarrow \tau \to \tau \qquad \Gamma \vdash e_v \Leftarrow \tau}{\Gamma \vdash \mathsf{Qn}[e_n]\ e_v \Rightarrow \mathsf{Q}\ e_n}\ (\textsc{Qn})
$$

$$
\frac{\Gamma \vdash e_n \Rightarrow \tau_1 \to \tau_1 \qquad \Gamma \vdash e_1 \Rightarrow \mathsf{Q}\ e_n \qquad \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_= \Leftarrow \mathsf{Eq}\ e_2\ (e_2[e_n\ x/x])}{\Gamma \vdash \mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \tau_2}\ (\textsc{Qout})
$$

$$
\mathsf{Qin}[e_n]\ e \rightsquigarrow \mathsf{Qn}[e_n]\ (e_n\ e) \qquad \mathsf{let}[e_=]\ \mathsf{Qin}\ x = \mathsf{Qn}[e_n]\ e_v\ \mathsf{in}\ e_b \rightsquigarrow e_b[e_v/x]
$$

Fig. 10. Quotient types.

The rule Qout first checks that e_1 is well typed, which also returns the type $\mathsf{Q}\ e_n$ from which we can extract τ_1 which we need to typecheck the body e_2. Finally we check $e_=$ which should guarantee that e_2 will not expose the differences between quotiented values that belong to the same equivalence class. We do that by requiring $e_=$ to prove that e_2 returns the same answer whether or not normalization has been performed, which is the property we need to justify why normalization is performed during type checking but not during run-time.

In return for this promise not to observe the differences hidden by the quotient, we get a stronger conversion rule thanks to the eager normalization performed by the reduction of Qin.

4.5 Properties

4.5.1 Properties of our target language. Some of the basic metatheoretic properties of the target language are easy to establish:

Lemma 4.2 (Term substitution).
Given a well-formed context $\Gamma = \Gamma_1, x : \tau_2, \Gamma_2$ *and* $\Gamma_1 \vdash e_2 \Rightarrow \tau_2$*:*
If $\Gamma \vdash e_1 \Rightarrow \tau_1$ *then* $\Gamma_1, \Gamma_2 \vdash e_1[e_2/x] \Rightarrow \tau_1[e_2/x]$ *and*
If $\Gamma \vdash e_1 \Leftarrow \tau_1$ *then* $\Gamma_1, \Gamma_2 \vdash e_1[e_2/x] \Leftarrow \tau_1[e_2/x]$.

Proof. By induction on the typing derivation of e_1. □

Lemma 4.3 (Level substitution).
Given a well-formed context Γ*:*
If $\Gamma \vdash e \Rightarrow \tau$ *then* $\Gamma[\ell/l] \vdash e[\ell/l] \Rightarrow \tau[\ell/l]$ *and*
If $\Gamma \vdash e \Leftarrow \tau$ *then* $\Gamma[\ell/l] \vdash e[\ell/l] \Leftarrow \tau[\ell/l]$

Proof. By induction on the typing derivation of e. □

$$
\begin{array}{lrcl}
\textit{(eterms)} & e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \langle e_1, ..., e_n\rangle \mid e.i \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
& & \mid & \mathcal{U} \mid \to \mid \langle \bullet \rangle \mid \exists \mid \mathsf{Eq} \mid \mathsf{Q} \mid \mathsf{refl} \\
\textit{(econtexts)} & E & ::= & \bullet \mid E_1\ e_2 \mid e_1\ E_2 \mid \langle \vec{e_b}, E, \vec{e_a}\rangle \mid E.i \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ e_2
\end{array}
$$

$\boxed{e_1 \rightsquigarrow e_2}$ e_1 reduces to e_2:

$$(\lambda x.e_1)\ e_2 \rightsquigarrow e_1[e_2/x] \qquad \langle e_1, ..., e_n\rangle.i \rightsquigarrow e_i$$

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow e_2[e_1/x] \qquad \frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']}$$

Fig. 11. The type-erased language

Lemma 4.4 (Subject reduction).
Given a well-formed context Γ*, if* $\Gamma \vdash e_1 \Leftarrow \tau$ *and* $e_1 \rightsquigarrow e_2$*, then* $\Gamma \vdash e_2 \Leftarrow \tau$.

Proof. By induction on $e_1 \rightsquigarrow e_2$. □

Whether the target language is strongly normalizing is currently unknown. Almost all the constructs in the language are sufficiently well-understood that we can confidently say that they do not threaten strong normalization, with the single exception of the U-∃ rule whose soundness has not been seriously investigated. We discuss this in more detail in Section 6.5.

Type checking can be shown easily to be decidable, since all the rules are designed to represent an algorithm, with the notable exception of the Red rules which are not syntax directed. Of course, the proof can only succeed under the assumption that the language is strongly normalizing.

Fig. 11 shows the language used as the target of the erasure operation $|\cdot|$. The first line of the terms shows this is simply an untyped lambda calculus with tuples and the rest are simply constants representing the type constructors of our target language.

Lemma 4.5 (Erasure soundness).
If $\bullet \vdash e_1 \Leftarrow \tau$ *and* $|e_1| \rightsquigarrow e_2$*, then*
there exists an e_3 *such that* $\bullet \vdash e_3 \Leftarrow \tau$ *and* $e_1 \simeq e_3$ *and* $|e_3| = e_2$.

Proof. By induction on $|e_1| \rightsquigarrow e_2$ and $\bullet \vdash e_1 \Leftarrow \tau$. The proof follows the same steps as in [Monnier 2024]: The crucial ingredient is that we consider only expressions in the empty context. We rely on canonical forms lemmas to show that if e_1 is one of the elimination forms that get erased to a let, then the argument has to be reducible to the matching constructor so we can apply the corresponding reduction rule.

We rely on the same canonical forms lemma in a more subtle form when e_1 is of the form $\mathsf{let}[e_=]\ \mathsf{Qin}\ x = \mathsf{Qin}[e_n]\ e_1\ \mathsf{in}\ e_2$ which before erasure reduces in two steps to $e_2[e_n\ e_1/x]$ whereas after erasure this reduces in one step to $|e_2|[|e_1|/x]$: The $e_=$ annotation gives us a proof that $e_2[e_n\ e_1/x]$ is propositionally equal to $e_2[e_1/x]$ and we use the canonical forms lemma to show that $e_=$ has to be reducible to refl which means that the two expressions are not just propositionally but definitionally equal. □

4.5.2 Properties of the closure conversion. Our closure conversion algorithm is designed like a syntactic model, following the approach advocated in [Boulier et al. 2017].

Lemma 4.6 (substitution commutes).
Given a well-formed source context $\Gamma = \Gamma_1, x : \tau_1, \Gamma_2$ *and* $\Gamma_1 \vdash e_2 \Rightarrow \tau_1$*:*
If $\Gamma \vdash e_1 \Rightarrow \tau_1$ *or* $\Gamma \vdash \tau_1 \Rightarrow s$ *and* $\Gamma \vdash e_1 \Leftarrow \tau_1$ *then* $[\![e_1[e_2/x]]\!] \simeq [\![e_1]\!]\ [[\![e_2]\!]/x]$.

Proof. By induction on e_1. When e_1 is not a lambda expression, the induction is straightforward. For the lambda case, the closure conversion of the lambda before and after substitution results in a quite different result since the set of free variables is changed. The proof hinges on the use of the Qed rule of the quotient type. It starts by using the induction hypothesis to show that both conversions will return quotients of the same type and hence using the same normalization function, and then applies on both sides the shared normalization function, which ends up undoing the closure conversion. □

Lemma 4.7 (Computational soundness).
Given a well-formed source context Γ, *if* $\Gamma \vdash e_1 \implies \tau$ *and* $\Gamma \vdash e_2 \implies \tau$ *and* $e_1 \simeq e_2$, *then* $[\![e_1]\!] \simeq [\![e_2]\!]$.

Proof. By induction on the derivation of $e_1 \simeq e_2$. The non-trivial case is the β-reduction rule, where we see again 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 4.8 (Typing soundness).
Given a well-formed source context Γ, *if* $\Gamma \vdash e \implies \tau$, *then* $[\![\Gamma]\!] \vdash [\![e]\!] \implies [\![\tau]\!]$.

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 follows the same general structure as the usual proof of type preservation of closure conversion, such as found in [Savary-Bélanger et al. 2015]. □

Lemma 4.9 (Closedness).
Given a well-formed source context Γ, *if* $\Gamma \vdash e \implies \tau$, *for all* λ*-expressions* $\lambda x.e'$ *in* $|[\![e]\!]|$ *we have that* $fv(\lambda x.e') = \emptyset$.

Proof. By induction on the typing derivation. Most of the $\lambda x.e$ that can occur in the converted code appear in type annotations and get stripped away by the type erasure, such as the $[\![\cdot]\!]_{\text{norm}}$ used in the quotient type of functions. The only $\lambda x.e$ that still appears in after erasure are the main ones, which gets stashed on the *code* slot of the closure, and these are indeed closed thanks to the let and letcast placed around the body to rebind its free variables. □

5 CLOSURE CONVERTING A BIGGER LANGUAGE

Our source language was purposefully very limited, so that we could focus on the important elements, 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 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:

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

Adding support for existential types is similarly simple:

$$\begin{aligned}
[\![\exists x : \tau_1.\tau_2]\!] &= \exists x : [\![\tau_1]\!] . [\![\tau_2]\!] \\
[\![<e_1; e_2>]\!] &= <[\![e_1]\!]; [\![e_2]\!]> \\
[\![\text{let } <x_1; x_2> = e_1 \text{ in } e_2]\!] &= \text{let } <x_1; x_2> = [\![e_1]\!] \text{ in } [\![e_2]\!]
\end{aligned}$$

5.1 Closing the loop

The main motivations for this work is to replace the custom constructs used until now for closure conversion with more generic ones. Of course, by their generic nature, it is desirable to support them also in the source language. And indeed we can!

We can easily extend our source language with the same ∃ 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]:

$$\begin{array}{lcl} [\![\exists l.\tau]\!] & = & \exists l.\,[\![\tau]\!] \\ [\![<\ell; e>]\!] & = & <\ell; [\![e]\!]> \\ [\![\mathsf{let}\ <l; x> = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ <l; x> = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \end{array}$$

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. This can be just as easily extended with ∀ quantification over universe levels.

Finally, we can extend our source language to be the same as our target language by adding the remaining equality and quotient types. But this hits a minor hurdle: some of the new constructs such as letcast or Q take arguments which are expected to be functions so when closure converting them we have to be careful not to convert their function arguments into closures. There are various ways to circumvent the problem, but the simplest is to η-expand them:

$$\begin{array}{lcl} [\![\mathsf{Eq}\ e_1\ e_2]\!] & = & \mathsf{Eq}\ [\![e_1]\!]\ [\![e_2]\!] \\ [\![\mathsf{refl}]\!] & = & \mathsf{refl} \\ [\![\mathsf{letcast}[e_m, e_=]\ x\ =\ e_1\ \mathsf{in}\ e_2\ ]\!] & = & \mathsf{letcast}[\lambda y.\,[\![e_m\ y]\!], [\![e_=]\!]]\ x\ =\ [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \\ [\![\mathsf{Q}\ e_n]\!] & = & \mathsf{Q}\ (\lambda x.\,[\![e_n\ x]\!]) \\ [\![\mathsf{Qin}[e_n]\ e_v]\!] & = & \mathsf{Qin}[\lambda x.\,[\![e_n\ x]\!]]\ [\![e_v]\!] \\ [\![\mathsf{let}[e_=]\ \mathsf{Qin}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}[\lambda x_1.\lambda x_2.\lambda x_=.\,[\![e_=\ x_1\ x_2\ x_=]\!]]\ \mathsf{Qin}\ x = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \end{array}$$

Note that this works only because those places where we need to keep using actual functions are all erased anyway, so it does not matter if the lambda-expression we generate for them are not closed. For constructs that take function arguments and whose arguments are *not* erased, we would need to find other solutions, like we did with letcast.

With these extensions, 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 Other attempts

There are very few attempts to implement a type preserving closure conversion of a dependently typed language in the literature.

The oldest mention we could find is in Monnier and Haguenauer [2010] where the authors do not actually present a closure conversion, but instead present a conversion from a dependently typed language to a language where the dependencies are encoded as singleton types, after which they argue that a more or less standard closure conversion can be used. It is not clear if that would really work.

The other one is the work by Bowman and Ahmed [2018], which shows not only the closure conversion algorithm but also proves that the target language is consistent.

In the vicinity, Bowman et al. [2018] shows how to perform a CPS conversion for a dependently typed language. It does not rely on custom constructs but does not support a universe hierarchy and relies on a Prop-style impredicativity which limits it to a single universe.

More recently Koronkevich et al. [2022] attacked the related problem of a conversion to A-normal form for a dependently typed language. As expected, this proves an easier target and they manage to handle a language with the full universe tower and without requiring any form of impredicativity.

6.2 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.

Based on past experience with type-preserving closure conversion in non-dependent settings, we do not expect any significant difficulty adding support for inductive types: we already support dependent tuples and equality types, so fundamentally all that is missing is sum types and recursive types, neither of which usually interacts in any way with closure conversion, where the type preservation proof proceeds directly by applying the induction hypothesis. We can of course expect some superficial obstacles like those that forced us to replace cast with letcast, and clearly there could be additional complications linked to the usual syntactic termination checks which will likely tend to get confused by the layers of tuples and letcast added by the closure conversion. At the same time, the range of difficulties introduced by dependent types during closure conversion cannot be overstated, so it is of course possible that some nasty surprises may be lurking there, despite our optimistic expectations. The same should hold for coinductive types.

In contrast, it is very much unclear how an impredicative Prop universe would interact with the rest of this language. Maybe the fact that terms from the Prop universe can be erased could save us from having to figure it out. 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.

6.3 Efficiency

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

- The main issue with our conversion is that after type erasure, our closures are represented as pairs of the captured environment and the code. 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 single tuple that contains the code in one field and the free variables in the other fields. Furthermore, the code receives as argument not just the environment, but the whole closure (since they do not exist separately any more), requiring a non-trivial form of self-application.
- A more subtle 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.

- Compilation time could also prove to be an issue: our representation of closures before type erasure is fairly verbose with many layers of wrapping which are meant to be erased but still cost resources during compilation up until the time we can perform the erasure. Whether this would prove significant in a real compiler is unclear at this stage, as are the mitigating measures one could take if needed.
- Another serious limitation of our conversion is that our closures capture the whole set of *transitively* free variables, even though at run time only the truly free variables are required. It should be possible to solve this problem by marking some of the fields in the tuples as erasable, but we have not investigated this yet.

On the other hand, it should be possible to adapt our conversion algorithm to other non-flat closure representations such as that of Shao [1997] without changing our target language. This kind of flexibility is one of the benefits of representing closure objects as normal tuples rather than custom constructs.

6.4 Quotient types

Our approach is not necessarily tied to the specific choice of quotient type we decided to use, but the more common presentations of higher inductive types and quotient types do not offer a strengthened definitional equality, making it necessary to manipulate propositional proofs of equality between closures. This is turn would require the same kind of efforts as required to convert code from ETT to ITT [Winterhalter et al. [n. d.]]. We see no reason to think it cannot be made to work, but we have not tried to work out the details.

Similarly, we have not really tried to make our approach work with other quotient types like those of Cohen [2013] or Courtieu [2001] which are also based on normalization functions but they do not seem to be directly usable because they presume that the normalization function will also be used at run-time, which we cannot afford.

6.5 Impredicative Universe Quantification

The majority of the features we add to our target language to support closure conversion can be argued to be not only generic, in the sense that they can have many other uses, but also fairly standard in the sense that they, or variants of them, are already studied in several other languages. There is one major exception: the rule we use to type our existential universe types.

While languages like Agda support universe polymorphism as well as existential quantification over universe levels, they do it in a predicative way, typically placing such a quantified type as $\exists l.\tau$ into a universe over which they cannot quantify, such as $\mathcal{U}_\omega$, regardless of τ. There is a fair bit of recent research around universe polymorphism, such as by Bezem et al. [2022] and Hou et al. [2023], but they stay within a clearly predicative world or at most accommodate a single Prop-style impredicative universe, whereas closure conversion cannot preserve types without *some form* of impredicativity in all universes.

In contrast to Agda, we place such a type into the universe $\mathcal{U}_{\ell[0/l]}$, making it impredicative. As explained in Section 3.2, this rule is the one that our closure conversion "suggested".

6.5.1 Threats to consistency. 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 this novel notion of impredicativity introduced by more aggressive rules for universe polymorphism.

Whether it makes sense, is an open question. Impredicativity is a well known source of inconsistency, which is even the original motivation for the invention of types by Bertrand Russell. So requiring a new form of impredicativity is potentially problematic. We have not yet been able to prove the relative consistency of this kind of impredicativity. It is even far from obvious how to go about doing it.

As mentioned in Section 4.4.3, this is a qualitatively different kind of impredicativity than the traditional one seen in Prop. In our calculus, the bottom universe 0 is no less predicative than the others.[2] More importantly, this is not about specific universes being impredicative or not. To some extent it can be compared to known forms of impredicativity, as was shown by Monnier and Bos [2019] who prove that a similar impredicative universe quantification rule, applied to $\forall$ rather than $\exists$, is able to encode System-F.

6.5.2 Alternatives. We do not know that our calculus is consistent 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].

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. For example, our encoding could make do with a simpler construct $\exists t.\tau$ which would be equivalent to $\exists l.\exists t:\mathcal{U}_l.\tau$ but without giving access to l.

While the rule we use may ultimately prove dangerous in general, we do believe it to give the correct result for the specific case where the existential type is the form we use because those existential types are equivalent to the closures they represent from our source language, which is known to be consistent and does not involve any impredicativity: since the l of an object of type $\exists l.\tau$ is erasable, we cannot extract l out of it, nor can we extract from it any type that belongs to $\mathcal{U}_l$ (such as the t of the inner existential), nor for that matter any value whose type belongs to $\mathcal{U}_l$ (such as the field *env*) or contains an 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.

6.5.3 Signs of consistency. 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 their original λ-expression occupied in the source code. Sadly, that does not guarantee that they are sound in general.

This said, we have attempted to encode known paradoxes such as that of Hurkens [1995] in our target language system, and so far those attempts have not borne fruits. To give an example, some of those paradoxes begin by defining something like an ordering, represented by a type of the form $\langle t:\mathcal{U}_\ell, <: t \to t \to \mathcal{U}, \ldots\rangle$ and then want to define the ordering of all orderings, which requires some form of impredicativity. To make use of our impredicativity the natural choice is to use a type like $\exists l.\langle t:\mathcal{U}_l, <: t \to t \to \mathcal{U}, \ldots\rangle$, but then we aren't able to define an ordering between elements of this type because we cannot extract much useful information out of it, for the same reasons as mentioned above: since universe levels are second class we cannot project them out of the existential, which then prevents us from projecting the field t because its type would require

[2] Actually, one could argue that our $\mathcal{U}_0$ is "more predicative" since, as we noted in that same section, $\exists l.\tau$ can belong to universe 0 only in trivial cases.

projecting l, and this in turn also prevents us from projecting anything that mentions l or t out of this structure. So while it is not known to be consistent, it is currently not known to be inconsistent either. Of course it might be just a reflection of our inexperience.

Our intuition as for why impredicative universe quantification *may* not be completely crazy is that the second-class status of universe levels makes universe polymorphic definitions enjoy a strong form 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 rare 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 $\ell[0/l]$.

7 CONCLUSION

We have shown a target language together with a closure conversion algorithm that is able to handle a source language with dependent types, a tower of universes, and all that without resorting to the use of custom-made constructs. We also sketched how to extend it with tuples and all the other features supported by the target language, including universe polymorphism.

The main ingredients to get that novel result are quotient types and impredicative universe polymorphism, the latter of which is an as-yet poorly understood feature that will require further study to find out if it is consistent, and if not, whether some weaker form can be found that preserves consistency while still allowing uses such as closure conversion.

REFERENCES

Henk P. Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (April 1991), 121–154. https://doi.org/10.1017/S0956796800020025

Bruno Bernardo. 2009. Towards an Implicit Calculus of Inductive Constructions. Extending the Implicit Calculus of Constructions with Union and Subset Types. In *International Conference on Theorem Proving in Higher-Order Logics (Lecture Notes in Computer Science, Vol. 5674)*. https://hal.inria.fr/inria-00432649

Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. 2022. Type Theory with Explicit Universe Polymorphism. In *Types for Proofs and Programs (Leibniz International Proceedings in Informatics (LIPIcs))*. https://doi.org/10.4230/LIPIcs.TYPES.2022.13

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Certified Programs and Proofs*. 182–194. https://doi.org/10.1145/3018610.3018620

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

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible. In *Symposium on Principles of Programming Languages*. ACM Press. https://doi.org/10.1145/3158110

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

Cyril Cohen. 2013. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving*. 213–228. https://doi.org/10.1007/978-3-642-39634-2_17

Pierre Courtieu. 2001. Normalized Types. In *Computer Science Logic*. 554–569.

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *Comput. Surveys* 54, 5 (May 2021). https://doi.org/10.1145/3450952

Robert Harper and Mark Lillibridge. 1994. A type-theoretic approach to higher-order modules with sharing. In *Symposium on Principles of Programming Languages*. 123–137. https://doi.org/10.1145/174675.176927

Kuen-Bang Hou, Carlo Angiuli, and Reed Mullanix. 2023. An Order-Theoretic Analysis of Universe Polymorphism. In *Symposium on Principles of Programming Languages*. ACM Press. https://doi.org/10.1145/3571250

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

Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J. Bowman. 2022. ANF preserves dependent types up to extensional equality. *Journal of Functional Programming* 32 (2022). https://doi.org/10.1017/S0956796822000090

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. 2024. Normalizable Types. In *Workshop on Type-Driven Development*. 29–36. https://doi.org/10.1145/3678000.3678203

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

Olivier Savary-Bélanger, Stefan Monnier, and Brigitte Pientka. 2015. Programming type-safe transformations using higher-order abstract syntax. *Journal of Formalized Reasoning* 8, 1 (2015), 49–91. https://doi.org/10.6092/issn.1972-5787/5122

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

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. https://arxiv.org/abs/1308.0729

Simon Thompson. 1990. Lawful functions and program verification in Miranda. *Science of Computer Programming* 13, 2-3 (1990), 181–218.

D.A. Turner. 1985. Miranda: a non-strict functional language with polymorphic types. In *International Conference on Functional Programming Languages and Computer Architecture*. 1–16.

Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. [n. d.]. Eliminating Reflection from Type Theory. In *CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. https://hal.science/hal-01849166

A PROOFS

Lemma 4.6 (substitution commutes).
Given a well-formed source context $\Gamma = \Gamma_1, x:\tau_1, \Gamma_2$ *and* $\Gamma_1 \vdash e_2 \implies \tau_1$*:*
If $\Gamma \vdash e_1 \implies \tau_1$ *or* $\Gamma \vdash \tau_1 \implies s$ *and* $\Gamma \vdash e_1 \impliedby \tau_1$ *then* $[\![e_1[e_2/x]]\!] \simeq [\![e_1]\!][[\![e_2]\!]/x]$.

Proof. By induction on the typing derivation of e_1.

• Case $e_1 = x_v$:
if $x \neq x_v$ then $[\![x_v]\!]\ [[\![e_2]\!]/x] = x_v[[\![e_2]\!]/x] = x_v = [\![x_v]\!] = [\![x_v[e_2/x]]\!]$
otherwise $[\![x]\!]\ [[\![e_2]\!]/x] = x[[\![e_2]\!]/x] = [\![e_2]\!] = [\![x[e_2/x]]\!]$

• Case $e_1 = \mathcal{U}_\ell$, trivial: $[\![\mathcal{U}_\ell]\!]\ [[\![e_2]\!]/x] = \mathcal{U}_\ell[[\![e_2]\!]/x] = \mathcal{U}_\ell = [\![\mathcal{U}_\ell]\!] = [\![\mathcal{U}_\ell[e_2/x]]\!]$

• Case $e_1 = (M : \tau_1)$, more specifically:

$$\frac{\Gamma \vdash \tau_1 \implies s \qquad \Gamma \vdash M \impliedby \tau}{\Gamma \vdash (M : \tau_1) \implies \tau_1}$$

By induction we have

$$[\![M[e_2/x]]\!] \simeq [\![M]\!][[\![e_2]\!]/x]$$

and

$$[\![\tau_1[e_2/x]]\!] \simeq [\![\tau_1]\!][[\![e_2]\!]/x]$$

And thus

$$\frac{\dfrac{\dfrac{[\![M[e_2/x]]\!] \simeq [\![M]\!][[\![e_2]\!]/x] \qquad [\![\tau_1[e_2/x]]\!] \simeq [\![\tau_1]\!][[\![e_2]\!]/x]}{([\![M[e_2/x]]\!] : [\![\tau_1[e_2/x]]\!]) \simeq ([\![M]\!][[\![e_2]\!]/x] : [\![\tau_1]\!][[\![e_2]\!]/x])}}{[\![(M[e_2/x] : \tau_1[e_2/x])]\!] \simeq ([\![M]\!] : [\![\tau_1]\!])[[\![e_2]\!]/x]}}{[\![(M : \tau_1)[e_2/x]]\!] \simeq [\![(M : \tau_1)]\!][[\![e_2]\!]/x]}$$

• Case $e_1 = e_f\ e_a$, more specifically:

$$\frac{\Gamma \vdash e_f \implies (x:\tau_a) \to \tau_r \qquad \Gamma \vdash e_a \impliedby \tau_a}{\Gamma \vdash e_f\ e_a \implies \tau_r[(e_a:\tau_a)/x]}$$

By induction we have

$$[\![e_f[e_2/x]]\!] \simeq [\![e_f]\!][[\![e_2]\!]/x]$$

and

$$[\![e_a[e_2/x]]\!] \simeq [\![e_a]\!][[\![e_2]\!]/x]$$

And thus

$$\frac{\dfrac{\dfrac{[\![e_f[e_2/x]]\!] \simeq [\![e_f]\!][[\![e_2]\!]/x] \qquad [\![e_a[e_2/x]]\!] \simeq [\![e_a]\!][[\![e_2]\!]/x]}{(\mathsf{let}[\mathsf{refl}]\ \mathsf{Qin}\ c = [\![e_f[e_2/x]]\!]\ \mathsf{in}\ \mathsf{call}\ c\ [\![e_a[e_2/x]]\!]) \simeq (\mathsf{let}[\mathsf{refl}]\ \mathsf{Qin}\ c = [\![e_f]\!][[\![e_2]\!]/x]\ \mathsf{in}\ \mathsf{call}\ c\ [\![e_a]\!][[\![e_2]\!]/x]}}{[\![(e_f[e_2/x]\ e_a[e_2/x])]\!] \simeq (\mathsf{let}[\mathsf{refl}]\ \mathsf{Qin}\ c = [\![e_f]\!]\ \mathsf{in}\ \mathsf{call}\ c\ [\![e_a]\!])[[\![e_2]\!]/x]}}{[\![(e_f\ e_a)[e_2/x]]\!] \simeq [\![e_f\ e_a]\!][[\![e_2]\!]/x]}$$

We assume here that c is appropriately renamed to avoid name capture.

- Case $e_1 = \lambda x_a.e$, more specifically:

$$\frac{\Gamma, x_a : \tau_a \vdash e \;\Leftarrow\; \tau_r}{\Gamma \vdash \lambda x_a.e \;\Leftarrow\; ((x_a : \tau_a) \to \tau_r)}$$

We also know that

$$\frac{\Gamma \vdash \tau_a \;\Rightarrow\; \mathcal{U}_{\ell_a} \qquad \Gamma, x_a : \tau_a \vdash \tau_r \;\Rightarrow\; \mathcal{U}_{\ell_r} \qquad \ell = \max(\ell_a, \ell_r)}{\Gamma \vdash (x_a : \tau_a) \to \tau_r \;\Rightarrow\; \mathcal{U}_{\ell}}$$

and thus

$$\frac{\Gamma_1, \Gamma_2, x_a : \tau_a[[\![e_2]\!]/x] \vdash e[[\![e_2]\!]/x] \;\Leftarrow\; \tau_r[[\![e_2]\!]/x]}{\Gamma_1, \Gamma_2 \vdash \lambda x_a.e[[\![e_2]\!]/x] \;\Leftarrow\; ((x_a : \tau_a[[\![e_2]\!]/x]) \to \tau_r[[\![e_2]\!]/x])}$$

By induction we have

$$[\![e[e_2/x]]\!] \simeq [\![e]\!][[\![e_2]\!]/x]$$

and

$$[\![\tau_a[e_2/x]]\!] \simeq [\![\tau_a]\!][[\![e_2]\!]/x]$$

and

$$[\![\tau_r[e_2/x]]\!] \simeq [\![\tau_r]\!][[\![e_2]\!]/x]$$

and

$$[\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!] \simeq [\![(x_a : \tau_a) \to \tau_r]\!][[\![e_2]\!]/x]$$

We're trying to prove $[\![\lambda x_a.e[e_2/x]]\!] \simeq [\![\lambda x_a.e]\!][[\![e_2]\!]/x]$.
For simplicity, in the closure conversion we presume that tfv just returns all the vars, so $\Gamma' = \Gamma$ and $\vec{x} = \mathrm{Dom}(\Gamma)$.
$[\![\lambda x_a.e[e_2/x]]\!]$ expands to:

$$\begin{array}{l}
\mathsf{Qin}[[\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\mathrm{norm}}] \\
\quad (<\ell; <\langle [\![\Gamma_1, \Gamma_2]\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . body \rangle >> : [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\mathrm{inner}}) \\
\text{where} \quad \Gamma_1, \Gamma_2 \vdash \langle \Gamma_1, \Gamma_2 \rangle \;\Rightarrow\; \mathcal{U}_\ell \qquad \text{Compute the universe level } \ell \\
\qquad \vec{x} \quad = \mathrm{Dom}(\Gamma_1, \Gamma_2) \\
\qquad f_m \quad = \lambda \langle \vec{x} \rangle . (x_a : [\![\tau_a[e_2/x]]\!]) \to [\![\tau_r[e_2/x]]\!] \qquad \text{The } \textit{motive} \text{ of the cast} \\
\qquad body = \mathsf{letcast}[f_m, x_=]\ x_a \ =\ x'_a \ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \langle \vec{x} \rangle = x_e \ \mathsf{in}\ [\![e[e_2/x]]\!]
\end{array}$$

The normalization argument passed to Qin reduces to:

$$\begin{array}{l}
\mathit{cnorm} \\
= [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\mathrm{norm}} \\
= (\lambda c. <1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ c\ x_a \rangle >> \\
\qquad : [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\mathrm{inner}} \to [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\mathrm{inner}})
\end{array}$$

so the whole Qin reduces to:

$$
\begin{array}{l}
\mathsf{Qin}[\mathit{cnorm}] \\
\quad (<\ell; <\langle [\![\Gamma_1, \Gamma_2]\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body} \rangle >> : [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}}) \\
= \{ \text{ Reduce Qin to Qn } \} \\
\mathsf{Qn}[\mathit{cnorm}] \\
\quad ((\lambda c. <1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ c\ x_a \rangle >>) \\
\quad\ (<\ell; <\langle [\![\Gamma_1, \Gamma_2]\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body} \rangle >> : [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}})) \\
= \{ \ \beta\text{-reduce } \} \\
\mathsf{Qn}[\mathit{cnorm}] \\
\quad (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call} \begin{pmatrix} <\ell; <\langle [\![\Gamma_1, \Gamma_2]\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body} \rangle >> \\ : [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}} \end{pmatrix} x_a \rangle >>) \\
= \{ \text{ Inline+reduce: call } c\ x = \mathsf{let}\ <l; <t; \langle \mathit{env}, \mathit{code} \rangle >> = c\ \mathsf{in}\ \mathit{code}\ \langle \mathit{env}, x, \mathsf{refl} \rangle \ \} \\
\mathsf{Qn}[\mathit{cnorm}]\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . ((\lambda \langle x_e, x'_a, x_= \rangle . \mathit{body})\ \langle \langle \vec{x} \rangle, x_a, \mathsf{refl} \rangle) \rangle >>) \\
= \{ \text{ Reduce further } \} \\
\mathsf{Qn}[\mathit{cnorm}]\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . (\mathit{body}[\langle \vec{x} \rangle / x_e, x_a / x'_a, \mathsf{refl}/x_=]) \rangle >>) \\
= \{ \text{ Expose } \mathit{body} \ \} \\
\mathsf{Qn}[\mathit{cnorm}] \\
\quad (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . (\begin{pmatrix} \mathsf{letcast}[f_m, x_=]\ x_a\ =\ x'_a\ \mathsf{in} \\ \mathsf{let}\ \langle \vec{x} \rangle = x_e\ \mathsf{in}\ [\![e[e_2/x]]\!] \end{pmatrix} [\langle \vec{x} \rangle / x_e, x_a / x'_a, \mathsf{refl}/x_=]) \rangle >>) \\
= \{ \text{ Substitute into } \mathit{body} \ \} \\
\mathsf{Qn}[\mathit{cnorm}]\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \begin{pmatrix} \mathsf{letcast}[f_m, \mathsf{refl}]\ x_a\ =\ x_a\ \mathsf{in} \\ \mathsf{let}\ \langle \vec{x} \rangle = \langle \vec{x} \rangle\ \mathsf{in}\ [\![e[e_2/x]]\!] \end{pmatrix} \rangle >>) \\
= \{ \text{ Reduce } \} \\
\mathsf{Qn}[\mathit{cnorm}]\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . [\![e[e_2/x]]\!] \rangle >>)
\end{array}
$$

Now for $[\![\lambda x_a.e]\!][[\![e_2]\!]/x]$, it expands to:

$$
\begin{array}{l}
(\mathsf{Qin}[[\![(x_a : \tau_a) \to \tau_r]\!]_{\text{norm}}] \\
\quad (<\ell; <\langle [\![\Gamma]\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body} \rangle >> : [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}}))[[\![e_2]\!]/x] \\
\text{where} \quad \Gamma \vdash \langle \Gamma \rangle \ \Rightarrow\ \mathcal{U}_\ell \qquad\qquad\qquad \text{Compute the universe level } \ell \\
\qquad\quad \vec{x} \quad = \ \mathsf{Dom}(\Gamma) \\
\qquad\quad f_m \quad = \ \lambda \langle \vec{x} \rangle . (x_a : [\![\tau_a]\!]) \to [\![\tau_r]\!] \qquad \text{The } \textit{motive} \text{ of the cast} \\
\qquad\quad \mathit{body} = \ \mathsf{letcast}[f_m, x_=]\ x_a\ =\ x'_a\ \mathsf{in} \\
\qquad\qquad\qquad\ \ \mathsf{let}\ \langle \vec{x} \rangle = x_e\ \mathsf{in}\ [\![e]\!]
\end{array}
$$

After distributing the substitution to both arguments of Qin, the normalization argument passed to Qin reduces to:

$$
\begin{array}{l}
\mathit{cnorm}' \\
= [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{norm}}\ [[\![e_2]\!]/x] \\
= (\lambda c. <1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ c\ x_a \rangle >> \\
\qquad : ([\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}} \to [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}})[[\![e_2]\!]/x])
\end{array}
$$

The whole Qin reduces as follows:

$$
\begin{array}{l}
\mathsf{Qin}[\mathit{cnorm}'] \\
\quad (<\ell; <\langle [\![\Gamma]\!] \rangle; \langle \langle \vec{x} \rangle [[\![e_2]\!]/x], \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body}[[\![e_2]\!]/x] \rangle >> : [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}}\ [[\![e_2]\!]/x]) \\
= \{\ \text{Reduce Qin to Qn}\ \} \\
\mathsf{Qn}[\mathit{cnorm}'] \\
\quad ((\lambda c.<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ c\ x_a \rangle >>) \\
\quad\ (<\ell; <\langle [\![\Gamma]\!] \rangle; \langle \langle \vec{x} \rangle [[\![e_2]\!]/x], \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body}[[\![e_2]\!]/x] \rangle >> : [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}}\ [[\![e_2]\!]/x])) \\
= \{\ \beta\text{-reduce}\ \} \\
\mathsf{Qn}[\mathit{cnorm}'] \\
\quad (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ \left(<\ell; <\langle [\![\Gamma]\!] \rangle; \langle \langle \vec{x} \rangle [[\![e_2]\!]/x], \lambda \langle x_e, x'_a, x_= \rangle . \mathit{body}[[\![e_2]\!]/x] \rangle >>\right)\ x_a \rangle >>) \\
= \{\ \text{Inline+reduce: call}\ c\ x = \mathsf{let}\ <l; <t; \langle \mathit{env}, \mathit{code} \rangle >> = c\ \mathsf{in}\ \mathit{code}\ \langle \mathit{env}, x, \mathsf{refl} \rangle\ \} \\
\mathsf{Qn}[\mathit{cnorm}']\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . ((\lambda \langle x_e, x'_a, x_= \rangle . \mathit{body}[[\![e_2]\!]/x])\ \langle \langle \vec{x} \rangle [[\![e_2]\!]/x], x_a, \mathsf{refl} \rangle) \rangle >>) \\
= \{\ \text{Reduce further}\ \} \\
\mathsf{Qn}[\mathit{cnorm}']\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . (\mathit{body}[[\![e_2]\!]/x][\langle \vec{x} \rangle [[\![e_2]\!]/x]/x_e, x_a/x'_a, \mathsf{refl}/x_=]) \rangle >>) \\
= \{\ \text{Swap the two substitutions}\ \} \\
\mathsf{Qn}[\mathit{cnorm}']\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . (\mathit{body}[\langle \vec{x} \rangle / x_e, x_a/x'_a, \mathsf{refl}/x_=][[\![e_2]\!]/x]) \rangle >>) \\
= \{\ \text{Expose}\ \mathit{body}\ \} \\
\mathsf{Qn}[\mathit{cnorm}'] \\
\quad (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \left(\begin{array}{l} \mathsf{letcast}[f_m, x_=]\ x_a\ =\ x'_a\ \mathsf{in} \\ \mathsf{let}\ \langle \vec{x} \rangle = x_e\ \mathsf{in}\ [\![e]\!] \end{array}\right) [\langle \vec{x} \rangle / x_e, x_a/x'_a, \mathsf{refl}/x_=][[\![e_2]\!]/x]) \rangle >>) \\
= \{\ \text{Substitute into}\ \mathit{body}\ \} \\
\mathsf{Qn}[\mathit{cnorm}']\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \left(\begin{array}{l} \mathsf{letcast}[f_m, \mathsf{refl}]\ x_a\ =\ x_a\ \mathsf{in} \\ \mathsf{let}\ \langle \vec{x} \rangle = \langle \vec{x} \rangle\ \mathsf{in}\ [\![e]\!] \end{array}\right) [[\![e_2]\!]/x] \rangle >>) \\
= \{\ \text{Reduce}\ \} \\
\mathsf{Qn}[\mathit{cnorm}']\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . [\![e]\!][[\![e_2]\!]/x] \rangle >>)
\end{array}
$$

Note that the $\vec{x}$ of captured variables in the two developments are not identical: in one it includes x and in the other it doesn't, but the Qin normalization successfully hides the difference.
So, now we need to show:

$$
\begin{array}{l}
\mathsf{Qn}[\mathit{cnorm}']\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . [\![e]\!][[\![e_2]\!]/x] \rangle >>) \\
\simeq \\
\mathsf{Qn}[\mathit{cnorm}]\ (<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . [\![e[e_2/x]]\!] \rangle >>)
\end{array}
$$

We already know that $[\![e]\!][[\![e_2]\!]/x] \simeq [\![e[e_2/x]]\!]$, so we only need to show $\mathit{cnorm} \simeq \mathit{cnorm}'$:

$$
\begin{array}{l}
\mathit{cnorm} = (\lambda c.<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ c\ x_a \rangle >> \\
\qquad\qquad : [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}} \to [\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}}) \\
\mathit{cnorm}' = (\lambda c.<1; <\langle \bullet \rangle; \langle \langle \rangle, \lambda \langle x_e, x_a, p \rangle . \mathsf{call}\ c\ x_a \rangle >> \\
\qquad\qquad : ([\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}} \to [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}})[[\![e_2]\!]/x])
\end{array}
$$

so we just need to show $[\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}} \simeq ([\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}})[[\![e_2]\!]/x]$:

$$
\begin{array}{l}
[\![(x_a : \tau_a[e_2/x]) \to \tau_r[e_2/x]]\!]_{\text{inner}} \\
= \{ \text{ by definition } \} \\
\exists l.\exists t : \mathcal{U}_l.\langle\ env\ :\ t, \\
\qquad\qquad code :\ \langle x_e : t, x : [\![\tau_a[e_2/x]]\!], p : (x_e = env)\rangle \to [\![\tau_r[e_2/x]]\!]\ \rangle \\
\simeq \{ \text{ by induction } \} \\
\exists l.\exists t : \mathcal{U}_l.\langle\ env\ :\ t, \\
\qquad\qquad code :\ \langle x_e : t, x : [\![\tau_a]\!]\ [[\![e_2]\!]/x], p : (x_e = env)\rangle \to [\![\tau_r]\!]\ [[\![e_2]\!]/x]\ \rangle \\
= \{ \text{ by hoisting\&consolidating the substitutions } \} \\
(\exists l.\exists t : \mathcal{U}_l.\langle\ env\ :\ t, \\
\qquad\qquad code :\ \langle x_e : t, x : [\![\tau_a]\!], p : (x_e = env)\rangle \to [\![\tau_r]\!]\ \rangle)[[\![e_2]\!]/x] \\
= \{ \text{ by definition } \} \\
([\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}})[[\![e_2]\!]/x]
\end{array}
$$

• Case $e_1 = (x_a : \tau_a) \to \tau_r$. This was already done as a subproblem of the lambda case.

□

Lemma 4.7 (Computational soundness).
Given a well-formed source context Γ, *if* $\Gamma \vdash e_1 \implies \tau$ *and* $\Gamma \vdash e_2 \implies \tau$ *and* $e_1 \simeq e_2$, *then* $[\![e_1]\!] \simeq [\![e_2]\!]$.

Proof. Since $e_1 \simeq e_2$ is defined in terms of $\rightsquigarrow$, the proof is really performed by proving that if $\Gamma \vdash e_1 \implies \tau$ and $\Gamma \vdash e_2 \implies \tau$ and $e_1 \rightsquigarrow e_2$, then $[\![e_1]\!] \simeq [\![e_2]\!]$, which proceeds by induction on the derivation of $e_1 \rightsquigarrow e_2$.

• Case $(N : \tau) \rightsquigarrow N$: trivial.

• Case $E[e] \rightsquigarrow E[e']$: By induction hypothesis, because inspection reveals that for all the possible source evaluation contexts E, the $[\![E[e]]\!]$ conversion always generates code of the form $E'[[\![e]\!]]$ for some target evaluation context E'.

• Case $((\lambda x.e_1) : (x : \tau_1) \to \tau_2)\ e_2 \rightsquigarrow (e_1 : \tau_2)[(e_2 : \tau_1)/x]$:

We need to show $[\![((\lambda x_a.e_1) : (x:\tau_a) \to \tau_r)\ e_2]\!] \simeq [\![(e_1 : \tau_r)[(e_2 : \tau_a)/x_a]]\!]$:

$[\![((\lambda x_a.e_1) : (x:\tau_a) \to \tau_r)\ e_2]\!]$
= {By definition of $[\![\cdot]\!]$ for applications}
let[refl] Qin $c = [\![((\lambda x_a.e_1) : (x:\tau_a) \to \tau_r)]\!]$ in call c $[\![e_2]\!]$
= {By definition of $[\![\cdot]\!]$ for type annotations}
let[refl] Qin $c = ([\![\lambda x_a.e_1]\!] : [\![(x_a:\tau_a) \to \tau_r]\!])$ in call c $[\![e_2]\!]$
= { By definition of $[\![\cdot]\!]$ for λ}
let[refl] Qin c = (Qin[$[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{norm}}$]
($<\ell; <\langle[\![\Gamma']\!]\rangle; \langle\langle\vec{x}\rangle, \lambda\langle x_e, x'_a, x_=\rangle.body\rangle>>$
: $[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{inner}}$)
: $[\![(x_a:\tau_a) \to \tau_r]\!]$)
in call c $[\![e_2]\!]$
$\rightsquigarrow^*$ { Reduce Qin to Qn }
let[refl] Qin c = (Qn[$[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{norm}}$]
($(\lambda c.<1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x_a, p\rangle.\text{call } c\ x_a\rangle>>$)
$<\ell; <\langle[\![\Gamma']\!]\rangle; \langle\langle\vec{x}\rangle, \lambda\langle x_e, x'_a, x_=\rangle.body\rangle>>$
: $[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{inner}}$)
: $[\![(x_a:\tau_a) \to \tau_r]\!]$)
in call c $[\![e_2]\!]$
$\rightsquigarrow^*$ { β-reduce, then inline+reduce call }
let[refl] Qin c = (Qn[$[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{norm}}$]
($<1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x_a, p\rangle.(\lambda\langle x_e, x'_a, x_=\rangle.body)\ \langle\langle\vec{x}\rangle, x_a, \text{refl}\rangle\rangle>>$
: $[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{inner}}$)
: $[\![(x_a:\tau_a) \to \tau_r]\!]$)
in call c $[\![e_2]\!]$
$\rightsquigarrow^*$ { Reduce the quotient then inline call }
let $<l; <t; \langle env, code\rangle>> = (<1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x_a, p\rangle.(\lambda\langle x_e, x'_a, x_=\rangle.body)\ \langle\langle\vec{x}\rangle, x_a, \text{refl}\rangle\rangle>>$
: $[\![(x_a:\tau_a) \to \tau_r]\!]_{\text{inner}}$)
in $code\ \langle env, [\![e_2]\!], \text{refl}\rangle$
$\rightsquigarrow^*$ { Reduce }
$(\lambda\langle x_e, x_a, p\rangle.(\lambda\langle x_e, x'_a, x_=\rangle.body)\ \langle\langle\vec{x}\rangle, x_a, \text{refl}\rangle)\ \langle\langle\rangle, [\![e_2]\!], \text{refl}\rangle$
$\rightsquigarrow^*$ { Reduce }
$(\lambda\langle x_e, x'_a, x_=\rangle.body)\ \langle\langle\vec{x}\rangle, [\![e_2]\!], \text{refl}\rangle$
$\rightsquigarrow^*$ { Expand $body$, β-reduce, and (re)display the type annotations }
letcast$[f_m, \text{refl}]\ x_a = ([\![e_2]\!] : [\![\tau_a]\!])$ in
let $\langle\vec{x}\rangle = \langle\vec{x}\rangle$ in $[\![e_1]\!]$
$\rightsquigarrow^*$ { Expand f_m }
letcast$[(\lambda\langle\vec{x}\rangle.(x_a:[\![\tau_a]\!]) \to [\![\tau_r]\!]), \text{refl}]\ x_a = ([\![e_2]\!] : [\![\tau_a]\!])$ in
let $\langle\vec{x}\rangle = \langle\vec{x}\rangle$ in $[\![e_1]\!]$
$\rightsquigarrow^*$ { Reduce the lets }
$([\![e_1]\!] : [\![\tau_r]\!])[([\![e_2]\!] : [\![\tau_a]\!])/x_a]$
= { By definition of $[\![\cdot]\!]$ for type annotations }
$[\![(e_1 : \tau_r)]\!]\ [[\![(e_2 : \tau_a)]\!]/x_a]$
$\simeq$ { By lemma 4.6 }
$[\![(e_1 : \tau_r)[(e_2 : \tau_a)/x_a]]\!]$

□

Lemma 4.8 (Typing soundness).
Given a well-formed source context Γ, *if* $\Gamma \vdash e \Rightarrow \tau$, *then* $[\![\Gamma]\!] \vdash [\![e]\!] \Rightarrow [\![\tau]\!]$.

Proof. The proof proceeds by induction on the typing derivation, with a strengthen induction where we also have $\Gamma \vdash e \Leftarrow \tau$, then $[\![\Gamma]\!] \vdash [\![e]\!] \Leftarrow [\![\tau]\!]$.

• Case x, more specifically:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau}$$

The proof is trivial:

$$\frac{\frac{[\![\Gamma]\!](x) = [\![\tau]\!]}{[\![\Gamma]\!] \vdash x \Rightarrow [\![\tau]\!]}}{[\![\Gamma]\!] \vdash [\![x]\!] \Rightarrow [\![\tau]\!]}$$

• Case $e_1\ e_2$, more specifically:

$$\frac{\Gamma \vdash e_1 \Rightarrow (x:\tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2[(e_2:\tau_1)/x]}$$

By induction we have

$$[\![\Gamma]\!] \vdash [\![e_1]\!] \Rightarrow [\![(x:\tau_1) \to \tau_2]\!]$$

and

$$[\![\Gamma]\!] \vdash [\![e_2]\!] \Leftarrow [\![\tau_1]\!]$$

First we can show that call $c\ [\![e_2]\!] : [\![\tau_2]\!]$:

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{[\![\Gamma]\!] \vdash [\![e_2]\!] \Leftarrow [\![\tau_1]\!]}{[\![\Gamma]\!], c:[\![(x:\tau_1) \to \tau_2]\!]_{\text{inner}}, t:\mathcal{U}_l, env:t, code:\ldots \vdash [\![e_2]\!] \Leftarrow [\![\tau_1]\!]}}{[\![\Gamma]\!], c:\ldots, t:\mathcal{U}_l, env:t, code:\ldots \vdash \langle env, [\![e_2]\!], \mathsf{refl}\rangle \Leftarrow \langle x_e:t, x:[\![\tau_1]\!], p:(x_e = env)\rangle}}{[\![\Gamma]\!], c:\ldots, t:\mathcal{U}_l, \ldots, env:t, code:\ldots \vdash code\ \langle env, [\![e_2]\!], \mathsf{refl}\rangle \Leftarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}}{[\![\Gamma]\!], c:\ldots, t:\mathcal{U}_l, c_2:\ldots \vdash \mathsf{let}\ \langle env, code\rangle = c_2\ \mathsf{in}\ code\ \langle env, [\![e_2]\!], \mathsf{refl}\rangle \Leftarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}}{[\![\Gamma]\!], c:\ldots, c_1:\ldots \vdash \mathsf{let}\ <t; \langle env, code\rangle> = c_1\ \mathsf{in}\ code\ \langle env, [\![e_2]\!], \mathsf{refl}\rangle \Leftarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}}{[\![\Gamma]\!], c:\ldots \vdash \mathsf{let}\ <l; <t; \langle env, code\rangle>> = c_1\ \mathsf{in}\ code\ \langle env, [\![e_2]\!], \mathsf{refl}\rangle \Leftarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}}{[\![\Gamma]\!], c:[\![(x:\tau_1) \to \tau_2]\!]_{\text{inner}} \vdash \mathsf{call}\ c\ [\![e_2]\!] \Leftarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}$$

Then we can show $\mathsf{refl} : \mathsf{Eq}\ (\mathsf{call}\ c\ [\![e_2]\!])\ (\mathsf{call}\ c\ [\![e_2]\!]\ [[\![(x:\tau_1) \to \tau_2]\!]_{\text{norm}}\ c/c])$ which we will need below in the code which extracts the closure from the quotient. The proof is not shown because it follows the same pattern as the reductions we've seen in the proof of lemma 4.7.
With that, we can conclude:

$$\frac{\frac{\frac{\frac{[\![\Gamma]\!], c:[\![(x:\tau_1) \to \tau_2]\!]_{\text{inner}} \vdash \mathsf{call}\ c\ [\![e_2]\!] \Leftarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x] \qquad \frac{[\![\Gamma]\!] \vdash [\![e_1]\!] \Rightarrow [\![(x:\tau_1) \to \tau_2]\!]}{[\![\Gamma]\!] \vdash [\![e_1]\!] \Rightarrow \mathsf{Q}\ [\![(x:\tau_1) \to \tau_2]\!]_{\text{no}}}}{[\![\Gamma]\!] \vdash \mathsf{let}[\mathsf{refl}]\ \mathsf{Q}in\ c = [\![e_1]\!]\ \mathsf{in}\ \mathsf{call}\ c\ [\![e_2]\!] \Rightarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}}{[\![\Gamma]\!] \vdash [\![e_1\ e_2]\!] \Rightarrow [\![\tau_2]\!][([\![e_2]\!] : [\![\tau_1]\!])/x]}}{[\![\Gamma]\!] \vdash [\![e_1\ e_2]\!] \Rightarrow [\![\tau_2]\!][[\![(e_2 : \tau_1)]\!]/x]}}{[\![\Gamma]\!] \vdash [\![e_1\ e_2]\!] \Rightarrow [\![\tau_2[(e_2 : \tau_1)/x]]\!]}$$

• Case $(x:\tau_1) \to \tau_2$, more specifically:

$$\frac{\Gamma \vdash \tau_1 \;\Rightarrow\; \mathcal{U}_{\ell_1} \qquad \Gamma, x:\tau_1 \vdash \tau_2 \;\Rightarrow\; \mathcal{U}_{\ell_2} \qquad \ell_3 = \max(\ell_1, \ell_2)}{\Gamma \vdash (x:\tau_1) \to \tau_2 \;\Rightarrow\; \mathcal{U}_{\ell_3}}$$

By the induction hypothesis we have:

$$[\![\Gamma]\!] \vdash [\![\tau_1]\!] \;\Rightarrow\; [\![\mathcal{U}_{\ell_1}]\!]$$

and

$$[\![\Gamma, x:\tau_1]\!] \vdash [\![\tau_2]\!] \;\Rightarrow\; [\![\mathcal{U}_{\ell_2}]\!]$$

We start by checking the type of $[\![(x:\tau_1) \to \tau_2]\!]_{\text{inner}}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\![\Gamma]\!] \vdash [\![\tau_1]\!] \;\Rightarrow\; [\![\mathcal{U}_{\ell_1}]\!]}{[\![\Gamma]\!], t:\mathcal{U}_l, env:t \vdash [\![\tau_1]\!] \;\Rightarrow\; \mathcal{U}_{\ell_1}} \qquad \dfrac{[\![\Gamma, x:\tau_1]\!] \vdash [\![\tau_2]\!] \;\Rightarrow\; [\![\mathcal{U}_{\ell_2}]\!]}{[\![\Gamma]\!], ..., x:[\![\tau_1]\!] \vdash [\![\tau_2]\!] \;\Rightarrow\; \mathcal{U}_{\ell_2}}}{[\![\Gamma]\!], t:\mathcal{U}_l, env:t \vdash \langle x_e:t, x:[\![\tau_1]\!], p:(x_e = env)\rangle \;\Rightarrow\; \mathcal{U}_{((1 \sqcup l) \sqcup \ell_1) \sqcup l}}}{[\![\Gamma]\!], t:\mathcal{U}_l, env:t \vdash \langle x_e:t, x:[\![\tau_1]\!], p:(x_e = env)\rangle \to [\![\tau_2]\!] \;\Rightarrow\; \mathcal{U}_{(((1 \sqcup l) \sqcup \ell_1) \sqcup l) \sqcup \ell_2}}}{[\![\Gamma]\!], t:\mathcal{U}_l \vdash \langle env:t, code:\langle x_e:t, x:[\![\tau_1]\!], p:(x_e = env)\rangle \to [\![\tau_2]\!]\rangle \;\Rightarrow\; \mathcal{U}_{l \sqcup ((((1 \sqcup l) \sqcup \ell_1) \sqcup l) \sqcup \ell_2)}}}{[\![\Gamma]\!] \vdash \exists t:\mathcal{U}_l.\langle env:t, code:...\rangle \;\Rightarrow\; \mathcal{U}_{(S\ l) \sqcup (l \sqcup ((((1 \sqcup l) \sqcup \ell_1) \sqcup l) \sqcup \ell_2))}}}{[\![\Gamma]\!] \vdash \exists l.\exists t:\mathcal{U}_l.\langle env:t, code:\langle x_e:t, x:[\![\tau_1]\!], p:(x_e = env)\rangle \to [\![\tau_2]\!]\rangle \;\Rightarrow\; \mathcal{U}_{\ell_3}}}{[\![\Gamma]\!] \vdash [\![(x:\tau_1) \to \tau_2]\!]_{\text{inner}} \;\Rightarrow\; \mathcal{U}_{\ell_3}}$$

The non-obvious step above is when the universe levels turn into ℓ_3 when the impredicative universe rule replaces l by 0, because:

$$\begin{array}{l}
(S\ l) \sqcup (l \sqcup ((((1 \sqcup l) \sqcup \ell_1) \sqcup l) \sqcup \ell_2))[0/l] \\
= \\
(S\ 0) \sqcup (0 \sqcup ((((1 \sqcup 0) \sqcup \ell_1) \sqcup 0) \sqcup \ell_2)) \\
\leadsto^* \\
1 \sqcup ((1 \sqcup \ell_1) \sqcup \ell_2) \\
\leadsto^* \{\ \ell_1 \text{ and } \ell_2 \text{ are constants} \geq 1\ \} \\
\max(\ell_1, \ell_2) \\
= \\
\ell_3
\end{array}$$

Finally we check the quotiented type:

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\llbracket\Gamma\rrbracket, c{:}..., ..., x{:}\llbracket\tau_1\rrbracket, ..., t{:}\mathcal{U}_l, ..., c_2{:}\langle env{:}t, code{:}...\rangle \vdash \langle c_2.1, x, \mathsf{refl}\rangle \Leftarrow \langle x_e{:}t, x{:}\llbracket\tau_1\rrbracket, p{:}(x_e = env)\rangle}{\llbracket\Gamma\rrbracket, c{:}..., ..., t{:}\mathcal{U}_l, ..., c_2{:}\langle env{:}t, code{:}\langle x_e{:}t, x{:}\llbracket\tau_1\rrbracket, p{:}(x_e = env)\rangle \to \llbracket\tau_2\rrbracket\rangle \vdash c_2.2\ \langle c_2.1, x, \mathsf{refl}\rangle \Rightarrow \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}..., ..., t{:}\mathcal{U}_l, c_2{:}\langle env{:}t, code{:}...\rangle \vdash \mathsf{let}\ \langle env, code\rangle = c_2\ \mathsf{in}\ code\ \langle env, x, \mathsf{refl}\rangle \Rightarrow \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}..., ..., c_1{:}\exists t{:}\mathcal{U}_l.\langle env{:}t, code{:}...\rangle \vdash \mathsf{let}\ <t; \langle env, code\rangle> = c_1\ \mathsf{in}\ code\ \langle env, x, \mathsf{refl}\rangle \Rightarrow \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}\exists l.\exists t{:}\mathcal{U}_l.\langle env{:}t, code{:}...\rangle, ... \vdash \mathsf{let}\ <l; <t; \langle env, code\rangle>> = c\ \mathsf{in}\ code\ \langle env, x, \mathsf{refl}\rangle \Rightarrow \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}\llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}}, ... \vdash \mathsf{let}\ <l; <t; \langle env, code\rangle>> = c\ \mathsf{in}\ code\ \langle env, x, \mathsf{refl}\rangle \Rightarrow \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}..., t{:}\mathcal{U}_1, env{:}\langle\bullet\rangle, x_e{:}\langle\bullet\rangle, x{:}\llbracket\tau_1\rrbracket, p{:}(x_e = env) \vdash \mathsf{call}\ c\ x \Rightarrow \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}..., t{:}\mathcal{U}_1, env{:}\langle\bullet\rangle \vdash \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x \Leftarrow \langle x_e{:}\langle\bullet\rangle, x{:}\llbracket\tau_1\rrbracket, p{:}(x_e = env)\rangle \to \llbracket\tau_2\rrbracket}}{\llbracket\Gamma\rrbracket, c{:}..., t{:}\mathcal{U}_1 \vdash \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x\rangle \Leftarrow \langle env{:}\langle\bullet\rangle, code{:}\langle x_e{:}\langle\bullet\rangle, x{:}\llbracket\tau_1\rrbracket, p{:}(x_e = env)\rangle \to \llbracket\tau_2\rrbracket\rangle}}{\llbracket\Gamma\rrbracket, c{:}... \vdash <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x\rangle> \Leftarrow \exists t{:}\mathcal{U}_1.\langle env{:}t, code{:}\langle x_e{:}t, x{:}\llbracket\tau_1\rrbracket, p{:}(x_e = env)\rangle \to \llbracket\tau_2\rrbracket\rangle}}{\llbracket\Gamma\rrbracket, c{:}\llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}} \vdash <1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x\rangle>> \Leftarrow \exists l.\exists t{:}\mathcal{U}_l.\langle env{:}t, code{:}...\rangle}}{\llbracket\Gamma\rrbracket, c{:}\llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}} \vdash <1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x\rangle>> \Leftarrow \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}}}}{\llbracket\Gamma\rrbracket \vdash \lambda c.<1; <\langle\bullet\rangle; \langle\langle\rangle, \lambda\langle x_e, x, p\rangle.\mathsf{call}\ c\ x\rangle>> \Leftarrow \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}} \to \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}}}}{\llbracket\Gamma\rrbracket \vdash \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{norm}} \Rightarrow \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}} \to \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{inner}}}}{\llbracket\Gamma\rrbracket \vdash \mathsf{Q}\ \llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{norm}} \Rightarrow \mathcal{U}_{\ell_3}}}{\llbracket\Gamma\rrbracket \vdash \llbracket(x{:}\tau_1) \to \tau_2\rrbracket \Rightarrow \llbracket\mathcal{U}_{\ell_3}\rrbracket}$$

• Case $\lambda x_a.e$, more specifically:

$$\frac{\Gamma, x_a{:}\tau_a \vdash e \Leftarrow \tau_r}{\Gamma \vdash \lambda x_a.e \Leftarrow ((x_a{:}\tau_a) \to \tau_r)}$$

By induction hypothesis we have:

$$\llbracket\Gamma\rrbracket, x_a{:}\llbracket\tau_1\rrbracket \vdash \llbracket e\rrbracket \Leftarrow \llbracket\tau_2\rrbracket$$

We already confirmed in the previous case that $\llbracket(x{:}\tau_1) \to \tau_2\rrbracket_{\text{norm}}$ is properly typed, so we don't check that again.
The main part is to check the body:

$$\frac{...}{\llbracket\Gamma\rrbracket, x_e{:}\langle\llbracket\Gamma'\rrbracket\rangle, x'_a{:}\llbracket\tau_a\rrbracket, x_={:}(x_e = \langle\vec{x}\rangle) \vdash \begin{array}{l}\mathsf{letcast}[f_m, x_=]\ x_a\ =\ x'_a\ \mathsf{in}\\ \mathsf{let}\ \langle\vec{x}\rangle = x_e\ \mathsf{in}\ \llbracket e\rrbracket\end{array} \Leftarrow \llbracket\tau_r\rrbracket}{\llbracket\Gamma\rrbracket, x_e{:}\langle\llbracket\Gamma'\rrbracket\rangle, x'_a{:}\llbracket\tau_a\rrbracket, x_={:}(x_e = \langle\vec{x}\rangle) \vdash body \Leftarrow \llbracket\tau_r\rrbracket}$$

So we need to check all the premises of the letcast typing rule. We have:

- $\llbracket\Gamma\rrbracket, ..., x_={:}(x_e = \langle\vec{x}\rangle) \vdash x_= \Rightarrow \mathsf{Eq}\ e_1\ e_2$
  Trivially true. This tells us that e_1 is x_e and e_2 is $\langle\vec{x}\rangle$ in those premises.
- $\llbracket\Gamma\rrbracket, ... \vdash f_m\ x_e \Rightarrow s$
  This check that f_m is a properly typed function returning a type. Since f_m is $\lambda\langle\vec{x}\rangle.(x_a : \llbracket\tau_a\rrbracket) \to \llbracket\tau_r\rrbracket$, we need $\llbracket\Gamma\rrbracket, ..., \llbracket\Gamma\rrbracket \vdash (x_a{:}\llbracket\tau_a\rrbracket) \to \llbracket\tau_r\rrbracket \Rightarrow s$ which we know already from the induction hypothesis.
- $f_m\ x_e \rightsquigarrow^* (x{:}\tau_{a1}) \to \tau_{r1}$
  This tells us that τ_{a1} is $\llbracket\tau_a\rrbracket[x_e.1/x_1, ...x_e.n/x_n]$ and τ_{r1} is similarly $\llbracket\tau_r\rrbracket[x_e.1/x_1, ...x_e.n/x_n]$.
- $f_m\ env \rightsquigarrow^* (x{:}\tau_{a2}) \to \tau_{r2}$
  This tells us that τ_{a2} is $\llbracket\tau_a\rrbracket[\langle\vec{x}\rangle.1/x_1, ...\langle\vec{x}\rangle.n/x_n]$, which is just $\llbracket\tau_a\rrbracket$, and τ_{r2} is $\llbracket\tau_r\rrbracket[vecx/vecx]$, thus $\llbracket\tau_r\rrbracket$.

- $[\![\Gamma]\!], ... \vdash x'_a \Leftarrow [\![\tau_a]\!]$
  Trivially true from the context.
- $[\![\Gamma]\!], ..., x_a : [\![\tau_a]\!][x_e.1/x_1, ...x_e.n/x_n] \vdash \text{let } \langle \vec{x} \rangle = x_e \text{ in } [\![e]\!] \Leftarrow [\![\tau_r]\!][x_e.1/x_1, ...x_e.n/x_n]$
  Let's get rid of the syntactic sugar:

$$[\![\Gamma]\!], x_e : \langle [\![\Gamma']\!] \rangle, ..., x_a : [\![\tau_a]\!][x_e.1/x_1, ...x_e.n/x_n] \vdash [\![e]\!][x_e.1/x_1, ...x_e.n/x_n] \Leftarrow [\![\tau_r]\!][x_e.1/x_1, ...x_e.n/x_n]$$

  This holds because we know by construction of Γ' that $x_e.n$ has the same type as x_n, and by induction we know that $[\![\Gamma]\!], x_a : [\![\tau_1]\!] \vdash [\![e]\!] \Leftarrow [\![\tau_2]\!]$, and then we get the above by repeated application of the substitution lemma.

Then we conclude by checking the overall code:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{By construction of } \vec{x}}{[\![\Gamma]\!] \vdash \langle \vec{x} \rangle \Leftarrow \langle [\![\Gamma']\!] \rangle} \quad \dfrac{[\![\Gamma]\!], x_e : \langle [\![\Gamma']\!] \rangle, x'_a : [\![\tau_a]\!], x_= : (x_e = \langle \vec{x} \rangle) \vdash body \Leftarrow [\![\tau_r]\!]}{[\![\Gamma]\!] \vdash \lambda \langle x_e, x'_a, x_= \rangle . body \Leftarrow \langle x_e : \langle [\![\Gamma']\!] \rangle, x : [\![\tau_a]\!], p : (x_e = \langle \vec{x} \rangle) \rangle \to [\![\tau_r]\!]}}{[\![\Gamma]\!] \vdash \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . body \rangle \Leftarrow \langle env : \langle [\![\Gamma']\!] \rangle, code : \langle x_e : \langle [\![\Gamma']\!] \rangle, x : [\![\tau_a]\!], p : (x_e = env) \rangle \to [\![\tau_r]\!] \rangle}}{[\![\Gamma]\!] \vdash <\langle [\![\Gamma']\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . body \rangle> \Leftarrow \exists t : \mathcal{U}_\ell . \langle env : t, code : \langle x_e : t, x : [\![\tau_a]\!], p : (x_e = env) \rangle \to [\![\tau_r]\!] \rangle}}{[\![\Gamma]\!] \vdash <\ell; <\langle [\![\Gamma']\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . body \rangle>> \Leftarrow \exists l . \exists t : \mathcal{U}_l . \langle \; env \; : \; t, \; code : \; \langle x_e : t, x : [\![\tau_a]\!], p : (x_e = env) \rangle \to [\![\tau_r]\!]}}{[\![\Gamma]\!] \vdash <\ell; <\langle [\![\Gamma']\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . body \rangle>> \Leftarrow [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}}}}{[\![\Gamma]\!] \vdash \begin{array}{l} \text{Qin}[[\![(x_a : \tau_a) \to \tau_r]\!]_{\text{norm}}] \\ (<\ell; <\langle [\![\Gamma']\!] \rangle; \langle \langle \vec{x} \rangle, \lambda \langle x_e, x'_a, x_= \rangle . body \rangle>> : [\![(x_a : \tau_a) \to \tau_r]\!]_{\text{inner}}) \end{array} \Rightarrow \text{Q } [\![(x : \tau_a) \to \tau_r]\!]_{\text{norm}}}}{[\![\Gamma]\!] \vdash [\![\lambda x_a . e]\!] \Rightarrow [\![(x_a : \tau_a) \to \tau_r]\!]}$$

• Case Conv, Red$_C$, *or* Red$_S$: For these cases, the closure conversion does not generate any code, and it is easy to see that it preserves typing by using the induction hypothesis together with the Computational soundness lemma 4.7. □