Typed Closure Conversion Preserves Observational Equivalence

Amal Ahmed Matthias Blume

Toyota Technological Institute at Chicago
{amal,blume}@tti-c.org

Abstract

Language-based security relies on the assumption that all potential attacks are bound by the rules of the language in question. When programs are compiled into a different language, this is true only if the translation process preserves observational equivalence.

We investigate the problem of fully abstract compilation, i.e., compilation that both *preserves* and *reflects* observational equivalence. In particular, we prove that typed closure conversion for the polymorphic λ-calculus with existential and recursive types is fully abstract. Our proof uses operational techniques in the form of a step-indexed logical relation and construction of certain *wrapper* terms that "back-translate" from target values to source values.

Although typed closure conversion has been assumed to be fully abstract, we are not aware of any previous result that actually proves this.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Security

Keywords equivalence-preserving compilation, full abstraction, step-indexed logical relations, typed closure conversion

1. Introduction

Large software systems consist of hundreds or thousands of components, and many of these components may be of uncertain origin. To ensure reliable and secure operation, it is important to defend against faulty or malicious code. Language-based security is built upon the concept of abstraction: if access to some private implementation detail might enable an attack, then this detail is made inaccessible by hiding it behind an abstract interface, for example using an existential type.

Results such as Reynold's *abstraction theorem* [26] provide the theoretical justification for this. Let $\mathcal{L}$ be a language and $P = C[A]$ be a program written in $\mathcal{L}$ where A is the implementation of an abstraction and C is its context, i.e., the "rest of the program" within which A is used. Given some other implementation A' of the same abstraction, the new program $P' = C[A']$ behaves identically to P. The intuition here is that the implementation A of the abstraction is *contextually equivalent* to the implementation A'.

To claim contextual equivalence one must consider the set of *all possible contexts*—i.e., the set of all the (well-typed) contexts that can be written down in language $\mathcal{L}$. This corresponds to the programmer's intuition, namely that using the abstraction facilities of $\mathcal{L}$ implies that any code interacting with the abstraction is bound by the rules of $\mathcal{L}$.

But what about contexts written in some *other* language? Compilers routinely translate programs from one language $\mathcal{L}$ to programs in another language $\mathcal{L}'$. Are programs that were contextually equivalent before translation (in $\mathcal{L}$) still contextually equivalent after translation (in $\mathcal{L}'$)? This is a security-relevant question that programmers *should* care about! If it is possible to mount an attack at the level of $\mathcal{L}'$, then the original reasoning is invalid—unless the translation *preserves* and *reflects* equivalences and is, therefore, *fully abstract*.

If the set of possible contexts in $\mathcal{L}'$ is restricted to those that can be obtained by translating $\mathcal{L}$-contexts, then the answer to the above question would be "yes," and the programmer's reasoning based on source language rules would be correct. Unfortunately, this is often not the case. For instance, Java programs [6] are often distributed in the form of bytecode for the Java Virtual Machine, and such code can be generated by means other than compiling Java source code; Microsoft's Common Language Runtime (CLR) was specifically designed to be the target of compilers for multiple languages [9]; most traditional compilers generate machine code, which can then be linked with other machine code. In all these situations it is easily possible that target contexts are *too powerful* in the sense that they can make observations that source contexts cannot. Indeed, Kennedy describes a number of ways in which abstractions were broken in the process of compiling $C^\sharp$ to the CLR intermediate language [14]. Similar problems with Java have previously been examined by Abadi [1].

As Kennedy points out, there are at least three approaches to repairing failures of full abstraction. First, we could enrich the source language itself so that every target-level observation has a source-level counterpart. But succeeding with this idea would be a rather questionable victory, since it merely amounts to weakening the abstraction facilities of the source language. The second approach is to weaken the target language to the point where, in some sense, it becomes merely an alternative notation for source programs, thereby guaranteeing that the only expressible target contexts trivially correspond to source contexts. In some specialized situations this might work, but it does not apply to foreign-function interfaces, plugin-architectures, or multi-lingual frameworks such as `.NET`.

The third—and most appealing—approach is to engineer the translation itself so that it uses target-level abstraction facilities in a clever enough fashion so that well-typed target contexts have no choice but to respect the original abstractions. Of course, this requires at least some abstraction facilities to be present in the target language. JVM bytecode and the CLR have been explicitly designed for this. Typed Assembly Language (TAL) [21] is one approach of bringing abstraction mechanisms even to low-level machine code.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

Assuming that the translation can be engineered in this way, the remaining problem is to show that the result really is fully abstract. While Kennedy describes solutions to those $C^\sharp$-to-CLR translation bugs he found, he also points out that we do not yet have a proof of the absence of other, still undiscovered bugs.

In this paper we examine closure conversion—a key step in the translation from System F to TAL [21]. We show that the translation scheme based on existential quantification over the type of the closure environment is, indeed, fully abstract and develop a method for proving this to be true in a language that has universally and existentially quantified types as well as recursive types.

The case of closure conversion. The basic idea of closure conversion is to collect the free variables of a λ-expression in a data structure (e.g., a record) called the *closure environment* and to pass this data structure as an additional argument to the λ-expression, thus turning it into a closed term. Typed closure conversion, which goes back to Minamide, Morrisett and Harper [20], holds the type of the closure environment abstract by existentially quantifying over it. In general, such existential quantification is necessary to make the translation term type-check at all. For instance, $\lambda x : \mathrm{int}.\, x + y$ and $\lambda x : \mathrm{int}.\, z\, x\, y$ have the same type (assuming the types of y and z are int and int → int → int, respectively), but their closure environments have different types given that in this case not even their lengths match. Typed closure conversion hides the difference behind an existential quantifier.

As we have discussed, full abstraction is at least as much a property of the translation as it is one of the target language. Therefore, a particular translation scheme can fail to be fully abstract even if the source- and target languages are identical. As we will see, keeping the closure environment completely under the existential quantifier has the desired property of making the translation fully abstract. However, it is not hard to craft variants of this "natural" translation that reveal intensional aspects of the closure, thus failing to be fully abstract. For example, the translation could expose the size of the closure environment as an additional non-abstract field of type int. Such a scheme might have benefits for data representation or downstream optimizers. But such a conversion also enables target contexts to distinguish between the λ-expressions shown above—even if z happens to be bound to a value such as $\lambda xy.\, x + y$ that renders the source terms equivalent—and thus is not fully abstract.

In this paper we consider the polymorphic call-by-value λ-calculus with existential and recursive types, and show that a typed closure conversion such as the one used by Morrisett et al. [21] is fully abstract. Though this has been believed to be true, we are not aware of any previous proof of this property. Parts of proofs elided in this paper can be found in the accompanying technical report [5].

2. Proving Translations Fully Abstract

Suppose we have a source language S equipped with a notion of contextual equivalence $\approx_{\mathrm{S}}$, and a target language T with its own notion of such an equivalence $\approx_{\mathrm{T}}$. Moreover, let there be a translation procedure $s \leadsto t$ that constructs a T-term t from a given S-term s.

Consider two terms and their translations: $s_1 \leadsto t_1$ and $s_2 \leadsto t_2$. We want $\leadsto$ to be *equivalence-preserving*—meaning that $s_1 \approx_{\mathrm{S}} s_2$ implies $t_1 \approx_{\mathrm{T}} t_2$, and *equivalence-reflecting*—meaning that $t_1 \approx_{\mathrm{T}} t_2$ implies $s_1 \approx_{\mathrm{S}} s_2$. Equivalence reflection captures the usual notion of *correctness* (i.e., *preservation of semantics* as in Leroy [15]): a translation is clearly not correct if it maps non-equivalent source terms to equivalent target terms. However, correct translations are often not equivalence-preserving. A translation is said to be *fully abstract* if it has both properties.

Because of the universal quantification over contexts, it is difficult to work with contextual equivalence directly. Therefore, the first step is to find a characterization of equivalence that is easier to work with. In this paper we will use logical relations [29, 25, 24], specifically, a *step-indexed* logical relation that is sound and complete with respect to contextual equivalence [4].

To show preservation of equivalence we need a way of making use of $s_1 \approx_{\mathrm{S}} s_2$ when showing $t_1 \approx_{\mathrm{T}} t_2$. The idea is to define a relation $s \propto t$ that—at the very least—holds whenever t is the result of translating s (i.e., $s \leadsto t$). If we can also prove that given $s \propto t$ we have $s' \propto t$ iff $s \approx_{\mathrm{S}} s'$ and $s \propto t'$ iff $t \approx_{\mathrm{T}} t'$ (that is, if we can show that $\propto$ respects $\approx_{\mathrm{S}}$ on the left and $\approx_{\mathrm{T}}$ on the right), then we can read the proof for full abstraction directly off the following commuting diagram:

$$\begin{tikzcd}
s_1 \arrow[r, leftrightarrow, "\approx_{\mathrm{S}}"] \arrow[d, "\propto"] & s_2 \arrow[d, "\propto"] \\
t_1 \arrow[r, leftrightarrow, "\approx_{\mathrm{T}}"] & t_2
\end{tikzcd}$$

The techniques for showing such properties (assuming they hold) clearly depend on details of S, T, and $\leadsto$. However, if we assume both S and T to contain λ-expressions (or λ-like expressions), a common problem is likely to surface: to prove that two λ-terms t_1 and t_2 in T are related, one has to show that they map any related arguments v_1^t and v_2^t to related results. The main fact to use here is that s_1 and s_2, i.e., the source-level λ-terms whose translations generated t_1 and t_2, are related. This means that s_1 and s_2 map any related *source* values v_1^s and v_2^s to related source results. How can we make use of this fact given that what we have are related *target* values, not source values? Given any v^t that can be an argument to a λ-term t, we must be able to show the existence of a v^s such that $v^s \propto v^t$. (In the opposite direction this would be easy, since we could simply translate v^s using $\leadsto$.)

The ability to "back-translate" any v^t of translation type—including those target values that are not in the image of the translation, possibly even containing subterms not of translation type—into a v^s seems crucial to proving that translations preserve equivalences. Roughly speaking, if we can back-translate values, then we can also back-translate entire contexts. (Think of the context as a function that is "applied" to the term in its hole.) Thus, to show that $s_1 \approx_{\mathrm{S}} s_2$ implies $t_1 \approx_{\mathrm{T}} t_2$ we can use proof by contradiction: a context C^t that distinguishes t_1 from t_2 can be back-translated to a context C^s that distinguishes s_1 from s_2, contradicting the premise. Therefore, the translation must have been equivalence-preserving.

2.1 Language choice

The back-translation problem does not disappear even when the translation is from S into S itself (i.e., when $T = S$), because in general the type of a translation term t does not match the type of its corresponding source term s. This is certainly the case for closure conversion, but also for other translations such as CPS transformation. However, if $S = T$, then—as we will see shortly (Section 2.2)—it is sometimes possible to characterize the relation $\propto$ as an isomorphism whose mediating coercion functions are *definable as terms of the language itself.*

Moreover, when $S \neq T$ it is often the case that T has a fairly straightforward one-to-one correspondence with a subset T' of S. Usually the target language has fewer high-level features, making the "forward" translation problem (from S to T) non-trivial since some source constructions have to be "translated away." The opposite direction (from T to S) is often much simpler due to the above-mentioned isomorphism between T and a subset of S. Thus, instead of considering the translation from S to T one can equivalently consider the corresponding translation from S to T'.

The situation is depicted in Figure 1. Consider the intended translation from source language S to target language T. Let there be a sub-language T' of S that is isomorphic to T. For the purpose

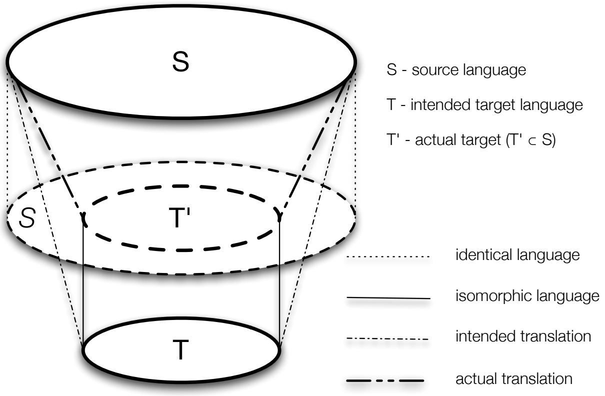

Figure 1. Languages

of comparing programs for contextual equivalence, it does not matter if one considers a term $t \in T$ or its counterpart $t' \in T'$. But every $t' \in T'$ is also a program in S, and any context C in T' that distinguishes a t'_1 from a t'_2 (both in T') is also a distinguishing context in S. Thus, if programs are not equivalent in T, then their counterparts in T' and S are also not equivalent, neither within just T' nor in S. Conversely, if we can prove that t'_1 is equivalent to t'_2 in S, then they are also equivalent within T'.[1]

Relying on this observation we will for the rest of the paper consider a direct translation from S to a subset of itself. The advantage of this approach is that we do not have to develop the machinery for reasoning about equivalence for two languages separately. Moreover, as hinted above and explained in much more detail below, it lets us characterize $\propto$ as a pair of coercion functions within S.

Notice, however, that equivalence in S is a *stronger* property and potentially more difficult to prove than equivalence in the targeted subset of S alone. At least for the case of closure conversion, this turns out not to be a problem and is outweighed by the above-mentioned advantages.

2.2 Wrapping

In this paper, we give a precise instance of our general proof outline for the case of closure conversion where S and T are the same *typed* language. Since terms contain types, the translation will have a type component that maps source types τ to corresponding target types τ^+. The *translation types* τ^+ form a strict subset of all possible target-level types.

We implement back-translation by giving a constructive method for generating source values from arbitrary target values (of translation type) and vice versa. Since S and T are the same language, we do so by using the language itself to define for each source type τ total functions $\mathcal{W}^-_{[\![\tau]\!]}$ (mapping from target to source) and $\mathcal{W}^+_{[\![\tau]\!]}$ (mapping from source to target). The terms $\mathcal{W}^\pm_{[\![\tau]\!]}$ are called *wrappers*. Similar wrappers have been used in many other settings, including contracts [11, 7], multi-language interoperability[2] [18], or representation analysis [16, 28].

$$
\begin{array}{llcl}
\textsf{Types} & \tau, \sigma & ::= & \alpha \mid \mathrm{int} \mid \forall[\bar{\alpha}](\bar{\tau}) \rightarrow \sigma \mid \exists \alpha.\tau \mid \\
 & & & \tau_1 \times \cdots \times \tau_n \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \\
\textsf{Sugar} & \tau_1 \rightarrow \tau_2 & = & \forall[](\tau_1) \rightarrow \tau_2 \\
 & \forall\alpha.\tau & = & \forall[\alpha]() \rightarrow \tau \\
\textsf{Contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha \\
 & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

$$
\begin{array}{lll}
\textsf{Exprs} \quad e & ::= & x \mid c \mid \lambda[\bar{\alpha}](\overline{x : \tau}).\, e \mid e_0[\bar{\tau}](\bar{e}) \mid \\
 & & \mathbf{pack}\,[\sigma, e]\,\mathbf{as}\,\exists\alpha.\tau \mid \mathbf{unpack}\,[\alpha, x] = e_1\,\mathbf{in}\,e_2 \mid \\
 & & (e_1, \ldots, e_n) \mid \pi_i(e) \mid \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \\
 & & (\mathbf{case}\,e_0\,\mathbf{of}\,\mathbf{inl}(x_1) \Rightarrow e_1 \mid \mathbf{inr}(x_2) \Rightarrow e_2) \mid \\
 & & \mathbf{fold}[\mu\alpha.\tau]e \mid \mathbf{unfold}\,e
\end{array}
$$

$$
\begin{array}{lrcll}
\textsf{Sugar} & \lambda x : \tau.\, e & = & \lambda[](x : \tau).\, e & \\
 & \lambda x.\, e & = & \lambda x : \tau.\, e & ;\ \tau \text{ inferred from context} \\
 & \textsf{id} & = & \lambda x.\, x & \\
 & e_1\, e_2 & = & e_1[](e_2) & \\
 & \mathbf{let}\,x = e_1\,\mathbf{in}\,e_2 & = & (\lambda x.\, e_2)\, e_1 & \\
 & e_1 \circ e_2 & = & (\lambda x_1.\, \lambda x_2.\, \lambda x.\, x_1(x_2(x)))\, e_1\, e_2 &
\end{array}
$$

$$
\begin{array}{l}
\mathbf{letrec}\,f : \tau \rightarrow \tau' = \lambda x.\, F\ \mathbf{and}\ g : \sigma \rightarrow \sigma' = \lambda y.\, G\ \mathbf{in}\ H \quad = \\
\quad \left\{ \begin{array}{l}
\mathbf{let}\ f^* = \lambda p : \tau_p.\, \lambda x : \tau. \\
\qquad \mathbf{let}\ u = \mathbf{unfold}\,p\ \mathbf{in}\ F[(\pi_1(u)\,p)/f, (\pi_2(u)\,p)/g] \\
\quad\ g^* = \lambda p : \tau_p.\, \lambda x : \sigma. \\
\qquad \mathbf{let}\ u = \mathbf{unfold}\,p\ \mathbf{in}\ G[(\pi_1(u)\,p)/f, (\pi_2(u)\,p)/g] \\
\quad\ p = \mathbf{fold}[\tau_p](f^*, g^*) \\
\ \mathbf{in}\ H[(f^*\,p)/f, (g^*\,p)/g]
\end{array} \right. \\
\quad \text{where } \tau_p = \mu\alpha.(\alpha \rightarrow \tau \rightarrow \tau') \times (\alpha \rightarrow \sigma \rightarrow \sigma')
\end{array}
$$

Figure 2. Syntax of types and expressions

Our central result is that a translation $s \rightsquigarrow t$ can be "faked" using wrappers; that is, we show that if $s \rightsquigarrow t$ then the result t of the translation is always equivalent to a suitably wrapped version $W[s]$ of the source term s (written $t \approx_\mathrm{T} W[s]$). This means that t and $W[s]$ behave identically in any target context C^t. Now, let $s_1 \rightsquigarrow t_1$ and $s_2 \rightsquigarrow t_2$. If $s_1 \approx_\mathrm{S} s_2$, then no context, including the source context $C^t[W[\cdot]]$, can distinguish between them. But $C^t[W[s_1]] \approx_\mathrm{T} C^t[t_1]$ and $C^t[W[s_2]] \approx_\mathrm{T} C^t[t_2]$, hence, $C^t[t_1] \approx_\mathrm{T} C^t[t_2]$, i.e., $t_1 \approx_\mathrm{T} t_2$.

The relation $\propto$, which we do not have to define separately for this style of proof, can be recovered using $\mathcal{W}^-$ and $\mathcal{W}^+$:

$$s \propto t \quad =_\mathrm{def} \quad s \approx_\mathrm{S} \mathcal{W}^-(t)$$

We will also be able to show that this is equivalent to $\mathcal{W}^+(s) \approx_\mathrm{T} t$.

3. Language

As we have explained, we use a single language in two roles, as both the source and the target of closure conversion. In addition to the technical advantages mentioned earlier, this also makes it easy to use closure conversion in a translation pipeline with other conversion steps, and provided these steps also preserve equivalences, the entire pipeline will be fully abstract. For example, one such other step is CPS-conversion. (While the basic idea behind a proof of full abstraction for CPS is the same, the details turn out to be substantially different. We plan to report on these separately.)

Types. The language is essentially the polymorphic λ-calculus (System F), with integers, products, sums, and existential as well as recursive types (see Figure 2). Unlike in most accounts of System F, however, we combine the universal quantifier $\forall$ and the function type constructor $\rightarrow$ into a single construct $\forall[\bar{\alpha}](\bar{\tau}) \rightarrow \tau_r$ which accommodates multiple formal type parameters[3] $\bar{\alpha} = \alpha_1, \ldots, \alpha_m$

[1] Of course, this form of reasoning cannot establish equivalence reflection, i.e., the preservation of non-equivalences. But for that it suffices to show how a distinguishing source-context can be translated into a distinguishing target context, which is usually easy given the general source-to-target translation mechanism.

[2] In essence, our wrappers provide interoperability between S and T.

[3] Throughout the paper we will use a line above a syntactic element as in $\bar{\alpha}$ or $\overline{x : \tau}$ to indicate lists of repeated instances of this element.

$$
\begin{array}{llcl}
\text{Values} & v & ::= & c \mid \lambda[\bar{\alpha}](\overline{x:\tau}).\, e \mid \mathbf{pack}\,[\sigma, v]\,\mathbf{as}\,\exists\alpha.\tau \mid \\
 & & & (v_1, \ldots, v_k) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid \mathbf{fold}[\mu\alpha.\tau]v \\
\text{Eval.} & E & ::= & [\cdot] \mid E[\bar{\tau}](\bar{e}) \mid v[\bar{\tau}](\bar{v}, E, \bar{e}) \mid \\
\text{contexts} & & & \mathbf{pack}\,[\sigma, E]\,\mathbf{as}\,\exists\alpha.\tau \mid \mathbf{unpack}\,[\alpha, x] = E\,\mathbf{in}\,e \mid \\
 & & & (\bar{v}, E, \bar{e}) \mid \pi_i(E) \mid \mathbf{inl}(E) \mid \mathbf{inr}(E) \mid \\
 & & & (\mathbf{case}\,E\,\mathbf{of}\,\mathbf{inl}(x_1) \Rightarrow e_1 \mid \mathbf{inr}(x_2) \Rightarrow e_2) \mid \\
 & & & \mathbf{fold}[\mu\alpha.\tau]E \mid \mathbf{unfold}\,E \\
\text{Step rule} & & & \dfrac{e \hookrightarrow e'}{E[e] \longrightarrow^1 E[e']}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Reductions} & (\lambda[\bar{\alpha}](\overline{x:\tau}).\, e)[\bar{\sigma}]\bar{v} & \hookrightarrow & e[\overline{\sigma/\alpha}][\overline{v/x}] \\
 & \mathbf{unpack}\,[\alpha, x] = \mathbf{pack}\,[\sigma, v]\,\mathbf{as}\,\exists\alpha.\tau\,\mathbf{in}\,e & \hookrightarrow & e[\sigma/\alpha][v/x] \\
 & \pi_i(v_1, \ldots, v_n) & \hookrightarrow & v_i \\
 & \mathbf{case}\,\mathbf{inl}(v)\,\mathbf{of}\,\mathbf{inl}(x_1) \Rightarrow e_1 \mid \mathbf{inr}(x_2) \Rightarrow e_2 & \hookrightarrow & e_1[v/x_1] \\
 & \mathbf{case}\,\mathbf{inr}(v)\,\mathbf{of}\,\mathbf{inl}(x_1) \Rightarrow e_1 \mid \mathbf{inr}(x_2) \Rightarrow e_2 & \hookrightarrow & e_2[v/x_2] \\
 & \mathbf{unfold}\,(\mathbf{fold}[\mu\alpha.\tau]v) & \hookrightarrow & v
\end{array}
$$

Figure 3. Operational semantics

as well as multiple value arguments of type(s) $\bar{\tau} = \tau_1, \ldots, \tau_n$. The difference is not so much fundamental as stylistic, since either approach can easily be simulated using the other. However, as we will see, a combined $\forall \rightarrow$ constructor makes it easy to express closure conversion as a translation from the language into itself, and it would also let us express CPS-conversion without the need to introduce more curried functions. Morrisett et al. [21] use the same approach in their "System F to TAL" work for the CPS- and closure-conversion target. A kinding judgment of the form $\Delta \vdash \tau$ states that τ is well-formed in context Δ. For brevity the usual rules for deriving such judgments are elided.

Expressions. The expression language (see Figure 2) includes variables and constants as well as introduction- and elimination forms for all the type constructors. In particular, $\lambda[\bar{\alpha}](\overline{x:\tau}).\, e$ is a polymorphic function with type parameters $\bar{\alpha} = \alpha_1, \ldots, \alpha_m$ and term parameters $\bar{x} = x_1, \ldots, x_n$ of types $\bar{\tau} = \tau_1, \ldots, \tau_n$. The corresponding elimination form is the combined type- and term application $e_0[\bar{\sigma}](\bar{e})$ where $\bar{\sigma} = \sigma_1, \ldots, \sigma_m$ is a list of types and $\bar{e} = e_1, \ldots, e_n$ is a list of expressions. As usual, the introduction form of $\exists\alpha.\tau$ is $\mathbf{pack}\,[\sigma, e]\,\mathbf{as}\,\exists\alpha.\tau$ where σ is the witness type and e is an expression of type $\tau[\sigma/\alpha]$. The corresponding elimination form is $\mathbf{unpack}\,[\alpha, x] = e_1\,\mathbf{in}\,e_2$ where α and x are type- and term variables, respectively, e_1 is an expression of type $\exists\alpha.\tau$, and e_2 is an expression that has access to the "unpacked" value of e_1 under the name x of type τ. We use an iso-recursive account of recursive types, using the $\mathbf{fold}$- and $\mathbf{unfold}$-forms as the mediating isomorphism between $\mu\alpha.\tau$ and $\tau[\mu\alpha.\tau/\alpha]$.

Typing judgments have the form $\Delta; \Gamma \vdash e : \tau$ where Δ is the kinding environment (listing free type variables of Γ, e and τ), and where Γ is the typing environment (assigning types to the free term variables of e). Figure 6 shows inference rules for *extended* typing judgments $\Delta; \Gamma \vdash e : \tau \rightsquigarrow e'$ that also show the closure conversion e' for each source term e. Rules for ordinary typing judgments can be obtained by eliding the $\rightsquigarrow e'$ part from these rules.

Operational semantics. In Figure 3 we give a conventional, call-by-value, small-step operational semantics for our language as a context-sensitive rewrite system in the style of Felleisen and Hieb [10].

Syntactic Sugar. Figure 2 also shows some syntactic sugar that we will use throughout the paper. Most of it can be seen as straightforward "macros." The definition of $\mathbf{letrec}$ is more involved, but ultimately just implements the well-known fixpoint solution for two mutually recursive functions. Using the techniques described in Section 4, it is relatively straightforward to justify a derived reduction rule for $\mathbf{letrec}$. Let us write L for the context $\mathbf{letrec}\ f : \tau \rightarrow \tau' = F\ \mathbf{and}\ g : \sigma \rightarrow \sigma' = G\ \mathbf{in}\ [\cdot]$. Then $L[H]$ and $H[L[f]/f, L[g]/g]$ are contextually equivalent.

4. A Step-Indexed Logical Relation

In this paper, we prove equivalence of programs using a slight variant of the step-indexed logical relation developed by the first author in prior work [4]. In this section, we briefly explain the essential elements of the construction.

The relational interpretation $\mathcal{V}\,[\![\tau]\!]$ of a closed type τ is a set of triples of the form (k, v_1, v_2) where k is a natural number (called the *approximation index* or *step index*), and v_1 and v_2 are (closed) values. Intuitively, $(k, v_1, v_2) \in \mathcal{V}\,[\![\tau]\!]$ says that in any computation running for no more than k steps, v_1 approximates v_2 at the type τ.

Preliminaries. A context C is an expression with a single hole $[\cdot]$ in it. Typing judgments for contexts have the form $(\Delta; \Gamma \vdash \tau) \rightrightarrows (\Delta'; \Gamma' \vdash \tau')$ where $(\Delta; \Gamma \vdash \tau)$ indicates the type of the hole (see context typing in Figure 4).

Definition 4.1 (contextual equivalence)
Let $\Delta; \Gamma \vdash e_1 : \tau$ *and* $\Delta; \Gamma \vdash e_2 : \tau$.

$$
\begin{array}{l}
\Delta; \Gamma \vdash e_1 \approx^{ctx} e_2 : \tau \quad =_{\text{def}} \\
\quad \forall C, \tau'.\ \vdash C : (\Delta; \Gamma \vdash \tau) \rightrightarrows (\cdot; \cdot \vdash \tau') \ \Rightarrow\ (C[e_1] \Downarrow \Leftrightarrow C[e_2] \Downarrow)
\end{array}
$$

Evaluation contexts E (Figure 3) are a subset of general contexts C. Since only closed terms can be placed in an evaluation context, the type of the hole always has the form $\cdot; \cdot \vdash \tau$.

In defining appropriate candidate sets—i.e., sets that can serve as valid interpretations of types (see below)—for the logical relation, we make use of the notion of *ciu-equivalence* (uses of closed instantiations) introduced by Mason and Talcott [17], which can be shown to be equivalent to contextual equivalence but is easier to work with since it cuts down the number of contexts under consideration. Here we only need to define ciu-equivalence for *closed* values — hence, what we are defining is just *use*-equivalence since the values are already closed. Two closed values v and v' of type τ are said to be *ciu-related* if, for any evaluation context E with a hole of type τ, if $E[v] \Downarrow$ then $E[v'] \Downarrow$ (Figure 4). Notice that ciu-relatedness is an approximate notion. Two values v and v' of type τ are *ciu-equivalent* if $v \preccurlyeq v' : \tau$ and $v' \preccurlyeq v : \tau$.

We use the meta-variable χ to denote sets of tuples of the form (k, v_1, v_2), where v_1 and v_2 are closed values (v_1,$v_2 \in$ CVal). We define candidate sets $\text{Type}_{\tau_1,\tau_2}$ (where τ_1 and τ_2 are closed types) as sets of those sets $\chi \subseteq \mathbb{N} \times \text{CVal} \times \text{CVal}$ that have the following three properties: if $(k, v_1, v_2) \in \chi$, then v_1 and v_2 must be well-typed with types τ_1 and τ_2 respectively; χ must be closed with respect to decreasing step-index; and χ must be *equivalence-respecting* (strictly speaking, approximation-respecting), which requires that if $(k, v_1, v_2) \in \chi$ and $v_2 \preccurlyeq v_2' : \tau$, then $(k, v_1, v_2') \in \chi$. This last requirement is needed for completeness of the logical relation with respect to contextual equivalence [4].

For any set χ such that for all $(k, v_1, v_2) \in \chi$ we have that $\cdot; \cdot \vdash v_2 : \tau$, we use the notation $\chi_\tau^{*\text{ciu}}$ to denote the transitive closure of χ under ciu-relatedness (see Figure 4).

Finally, we use the meta-variable ρ to denote type substitutions. These are partial maps from type variables α to triples (τ_1, τ_2, χ) where τ_1 and τ_2 are closed types. We define abbreviations for projecting the different components of the triple in Figure 4. In the next section, we give the interpretation of open types τ. These interpretations are parametrized by a type substitution ρ such that $\text{FTV}(\tau) \subseteq \text{dom}(\rho)$. We note that our interpretations ensure that if $\rho(\alpha) = (\tau_1, \tau_2, \chi)$ then $\chi \in \text{Type}_{\tau_1,\tau_2}$.

$$\begin{array}{ll}
\text{context typing:} & \vdash C : (\Delta;\Gamma \vdash \tau) \rightrightarrows (\Delta';\Gamma' \vdash \tau') \;\; =_{\text{def}} \;\; \forall e.\, \Delta;\Gamma \vdash e : \tau \;\Rightarrow\; \Delta';\Gamma' \vdash C[e] : \tau' \\
\text{ciu-relatedness:} & v \preccurlyeq v' : \tau \;\; =_{\text{def}} \;\; \forall E, \tau_1.\, (\vdash E : (\cdot;\cdot \vdash \tau) \rightrightarrows (\cdot;\cdot \vdash \tau_1) \;\wedge\; E[v] \Downarrow) \;\Rightarrow\; E[v'] \Downarrow \\
\text{candidate relations:} & \text{Type}_{\tau_1,\tau_2} \;=\; \{\, \chi \subseteq \mathbb{N} \times \text{CVal} \times \text{CVal} \mid \forall (j,v,v') \in \chi.\;\; \vdash v : \tau_1 \wedge \vdash v' : \tau_2 \;\wedge \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall i < j.\, (i,v,v') \in \chi \;\wedge \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall v''.\, v' \preccurlyeq v'' : \tau_2 \;\Rightarrow\; (j,v,v'') \in \chi \,\} \\
\text{ciu-closure:} & \chi_\tau^{*\text{ciu}} \;\; =_{\text{def}} \;\; \{(k,v,v'') \mid (k,v,v') \in \chi \;\wedge\; v' \preccurlyeq v'' : \tau\} \\
\text{rho components:} & \text{Let } \rho(\alpha) = (\tau_1, \tau_2, \chi)\text{. Then } \rho_1(\alpha) =_{\text{def}} \tau_1,\; \rho_2(\alpha) =_{\text{def}} \tau_2,\; \rho^{\text{rel}}(\alpha) =_{\text{def}} \chi.
\end{array}$$

Figure 4. Auxiliary definitions for the logical relation

$$\begin{array}{rcl}
\mathcal{V}[\![\alpha]\!]\,\rho & = & \rho^{\text{rel}}(\alpha) \\
\mathcal{V}[\![\text{int}]\!]\,\rho & = & \{\, (k,i,i) \mid i \in \mathbb{Z} \,\} \\
\mathcal{V}_\mu\,[\![k', \mu\alpha.\tau]\!]\,\rho & = & \{\, (k, \mathbf{fold}[\mu\alpha.\rho_1(\tau)]v_1, \mathbf{fold}[\mu\alpha.\rho_2(\tau)]v_2) \mid \\
 & & \qquad k \leq k' \;\wedge\; \forall j < k.(j, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\,\rho[\alpha \mapsto (\mu\alpha.\rho_1(\tau), \mu\alpha.\rho_2(\tau), \chi)] \quad \text{where } \chi = \mathcal{V}_\mu\,[\![j, \mu\alpha.\tau]\!]\,\rho \,\} \\
\mathcal{V}[\![\mu\alpha.\tau]\!]\,\rho & = & \bigcup\limits_{k' \geq 0} \mathcal{V}_\mu\,[\![k', \mu\alpha.\tau]\!]\,\rho \\
\mathcal{V}[\![\tau_1 \times \cdots \times \tau_n]\!]\,\rho & = & \{\, (k, (v_{11}, \ldots, v_{1n}), (v_{21}, \ldots, v_{2n})) \mid \forall i \in \{1, \ldots, n\}.\, (k, v_{1i}, v_{2i}) \in \mathcal{V}[\![\tau_i]\!]\,\rho \,\} \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]\,\rho & = & \{\, (k, \mathbf{inl}(v_1), \mathbf{inl}(v_2)) \mid (k, v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]\,\rho \,\} \cup \\
 & & \{\, (k, \mathbf{inr}(v_1), \mathbf{inr}(v_2)) \mid (k, v_1, v_2) \in \mathcal{V}[\![\tau_2]\!]\,\rho \,\} \\
\mathcal{V}[\![\forall[\bar{\alpha}](\bar{\tau}) \rightarrow \tau_r]\!]\,\rho & = & \{\, (k, \lambda[\bar{\alpha}](\overline{x : \rho_1(\tau)}).\, e_1, \lambda[\bar{\alpha}](\overline{x : \rho_2(\tau)}).\, e_2) \mid \\
\bar{\alpha} = \alpha_1, \ldots, \alpha_m & & \qquad \forall j < k.\, \forall p \in \{1, \ldots, m\}.\, \forall \sigma_{1p}, \sigma_{2p}.\, \forall \chi_p \in \text{Type}_{\sigma_{1p},\sigma_{2p}}. \\
\bar{\tau} = \tau_1, \ldots, \tau_n & & \qquad\qquad \text{let } \rho' = \rho[\alpha_1 \mapsto (\sigma_{11}, \sigma_{21}, \chi_1), \ldots, \alpha_m \mapsto (\sigma_{1m}, \sigma_{2m}, \chi_m)] \\
 & & \qquad\qquad\quad \text{if } \forall q \in \{1, \ldots, n\}.\, (j, v_{1q}, v_{2q}) \in \mathcal{V}[\![\tau_q]\!]\,\rho' \\
 & & \qquad\qquad\quad \text{then } (j, e_1[\overline{\sigma_1/\alpha}][\overline{v_1/x}], e_2[\overline{\sigma_2/\alpha}][\overline{v_2/x}]) \in \mathcal{C}[\![\tau_r]\!]\,\rho' \,\} \\
\mathcal{V}[\![\exists\alpha.\tau]\!]\,\rho & = & \{\, (k, \mathbf{pack}\,[\tau_1, v_1]\,\mathbf{as}\,\exists\alpha.\rho_1(\tau), \mathbf{pack}\,[\tau_2, v_2]\,\mathbf{as}\,\exists\alpha.\rho_2(\tau)) \mid \\
 & & \qquad \forall j \leq k.\, \forall f_1, f_2, \tau_0.\, \text{if } \text{dom}(\rho) \vdash \tau_0 \;\wedge\; (j, f_1, f_2) \in \mathcal{V}[\![\forall[\alpha](\tau) \rightarrow \tau_0]\!]\,\rho \\
 & & \qquad\qquad\qquad\qquad\qquad \text{then } (j, f_1[\tau_1]v_1, f_2[\tau_2]v_2) \in \mathcal{C}[\![\tau_0]\!]\,\rho \,\} \\
\mathcal{C}[\![\tau]\!]\,\rho & = & \{\, (k, e_1, e_2) \mid \forall j < k.\, \forall v_1.\, \text{if } e_1 \longrightarrow^j v_1 \text{ then } \exists v_2.\, e_2 \longrightarrow^* v_2 \;\wedge\; (k - j, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\,\rho \,\}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{D}[\![\cdot]\!] & = & \{\, \emptyset \,\} \\
\mathcal{D}[\![\Delta, \alpha]\!] & = & \{\, \rho[\alpha \mapsto (\tau_1, \tau_2, \chi) \mid \rho \in \mathcal{D}[\![\Delta]\!] \;\wedge\; \chi \in \text{Type}_{\tau_1,\tau_2} \,\} \\
\mathcal{G}[\![\cdot]\!]\,\rho & = & \{\, (k, \emptyset, \emptyset) \,\} \\
\mathcal{G}[\![\Gamma, x : \tau]\!]\,\rho & = & \{\, (k, \gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (k, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]\,\rho \;\wedge\; (k, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\,\rho \,\} \\
\Delta;\Gamma \vdash e_1 \leq e_2 : \tau & =_{\text{def}} & \Delta;\Gamma \vdash e_1 : \tau \;\wedge\; \Delta;\Gamma \vdash e_2 : \tau \;\wedge \\
 & & \forall k \geq 0.\, \forall \rho \in \mathcal{D}[\![\Delta]\!]\,.\, \forall (k, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]\,\rho.\, (k, \rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{C}[\![\tau]\!]\,\rho \\
\Delta;\Gamma \vdash e_1 \approx e_2 : \tau & =_{\text{def}} & \Delta;\Gamma \vdash e_1 \leq e_2 : \tau \;\wedge\; \Delta;\Gamma \vdash e_2 \leq e_1 : \tau
\end{array}$$

Figure 5. Step-indexed logical relation

Value relations. Figure 5 gives the relational interpretations of types $\mathcal{V}[\![\tau]\!]\,\rho$. This is a set of triples (k, v_1, v_2) where $\cdot;\cdot \vdash v_1 : \rho_1(\tau)$ and $\cdot;\cdot \vdash v_2 : \rho_2(\tau)$. (We elide these typing requirements from the definitions in Figure 5.)

Two values v_1 and v_2 are related at the type α for k steps, if $(k, v_1, v_2) \in \rho^{\text{rel}}(\alpha)$. Two integers are related at the type int for any number of steps if they are equal.

The values $\mathbf{fold}[]v_1$ and $\mathbf{fold}[]v_2$ (eliding types for the moment) are related at the type $\mu\alpha.\tau$ for k steps if v_1 and v_2 are related at the type $\tau[\mu\alpha.\tau/\alpha]$ for $k-1$ steps (or equivalently, for all $j < k$ steps). This requirement can equivalently be written as $(k-1, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\,\rho[\alpha \mapsto (\mu\alpha.\rho_1(\tau), \mu\alpha.\rho_2(\tau), \chi)]$ where χ is the set of all values related at the type $\mu\alpha.\tau$ for $i < k$ steps. Thus, notice that to determine if $(k, -, -) \in \mathcal{V}[\![\mu\alpha.\tau]\!]\,\rho$, we require a χ that contains all tuples $(i, -, -) \in \mathcal{V}[\![\mu\alpha.\tau]\!]\,\rho$ whose index i is *strictly less than* k. Hence our interpretation of recursive types is well-founded (as the formal definition in Figure 5 makes clear).

Functions are suspended computations that take both type and value arguments. They are related if for any suitable instantiation to type and value arguments their bodies are related as computations at the result type τ_r (under the augmented type substitution ρ' that takes type arguments into account). To be related for k steps, the functions' bodies must be related for the remaining $j < k$ steps at any time in the future when the function might be applied. Since function application itself takes up one step, $j < k$ (as opposed to $j \leq k$) suffices. A "suitable" type instantiation consists of any two types σ_1 and σ_2, together with an arbitrary relation χ drawn from $\text{Type}_{\sigma_1,\sigma_2}$. A suitable value instantiation picks arguments that are themselves related at type τ_1 (also under ρ') for the required j steps.

The interpretation of existential types is defined in terms of the interpretation of universal types. Intuitively, one can think of this as a "semantic" use of the dual encoding of existentials in terms of universals. To show that the values $\mathbf{pack}\,[\tau_1, v_1]\,\mathbf{as}\,\exists\alpha.\rho_1(\tau)$ and $\mathbf{pack}\,[\tau_2, v_2]\,\mathbf{as}\,\exists\alpha.\rho_2(\tau)$ are related for k steps at the type $\exists\alpha.\tau$ we proceed as follows. We pick two arbitrary functions $f_1 = \lambda[\alpha](x : \tau).\, e_1$ and $f_2 = \lambda[\alpha](x : \tau).\, e_2$ such that f_1 and f_2 are related for $j \leq k$ steps at the type $\forall[\alpha](\tau) \rightarrow \tau_0$ where $\alpha \notin \text{FTV}(\tau_0)$. (Thus, f_1 and f_2, like the body of an **unpack**, behave parametrically with respect to α.) We must show that $e_1[\tau_1/\alpha][v_1/x]$ and $e_2[\tau_2/\alpha][v_2/x]$ are related for $j-1$ steps as computations of type τ_0 (or equivalently: $(j, f_1[\tau_1]v_1, f_2[\tau_2]v_2) \in \mathcal{C}\llbracket\tau_0\rrbracket\,\rho$). Intuitively, this definition is just trying to capture how the elim form for existentials (i.e., **unpack**) works.

It may not be readily obvious that this interpretation of existential types is well-founded. One concern may be that we have picked an arbitrary type τ_0. Since our language has impredicative quantified types, τ_0 may be $\exists\alpha.\tau$ itself, i.e., the type whose interpretation we are trying to define! (In fact, τ_0 may even be a larger type than $\exists\alpha.\tau$.) As in the recursive types case, induction on the step-index (and not just on types) is crucial here for well-foundedness. But this leads to a second concern: the definition requires only that $j \leq k$ (rather than $j < k$). To see that the definition is well-founded regardless, notice that it is equivalent to the following (more verbose, but clearly well-founded) definition, which we get by "macro-expansion" of the clause $(j, -, -) \in \mathcal{V}\llbracket\forall[\alpha](\tau)\to\tau_0\rrbracket\,\rho$, and by replacing the clause $(j, -, -) \in \mathcal{C}\llbracket\tau_0\rrbracket\,\rho$ with an equivalent form:

$$\begin{array}{l}
\forall j \leq k.\, \forall f_1, f_2, \tau_0. \\
\quad \text{if } f_1 = \lambda[\alpha](x:\tau).\, e_1 \;\wedge\; f_2 = \lambda[\alpha](x:\tau).\, e_2 \;\wedge \\
\qquad \mathrm{dom}(\rho) \vdash \tau_0 \;\wedge \\
\qquad (\,\forall i < j.\, \forall \sigma_1, \sigma_2, \chi \in \mathrm{Type}_{\sigma_1,\sigma_2}. \\
\qquad\quad \text{let } \rho' = \rho[\alpha \mapsto (\sigma_1, \sigma_2, \chi)] \\
\qquad\qquad \forall (i, v_{11}, v_{22}) \in \mathcal{V}\llbracket\tau\rrbracket\,\rho'. \\
\qquad\qquad\quad (i, e_1[\sigma_1/\alpha][v_{11}/x], e_2[\sigma_2/\alpha][v_{22}/x]) \in \mathcal{C}\llbracket\tau_0\rrbracket\,\rho'\,) \\
\quad \text{then } (j-1, e_1[\tau_1/\alpha][v_1/x], e_2[\tau_2/\alpha][v_2/x]) \in \mathcal{C}\llbracket\tau_0\rrbracket\,\rho\,\}
\end{array}$$

Intuitively, the last line above is equivalent to $(j, f_1[\tau_1]v_1, f_2[\tau_2]v_2) \in \mathcal{C}\llbracket\tau_0\rrbracket\,\rho$ since beta-reduction consumes a step.

The interpretation of existential types that we use here is different from an earlier account [4] which, we discovered, did not satisfy the equivalence-respecting property mentioned above (see [3] for details). As a result, that logical relation was incomplete with respect to contextual equivalence at existential types. (The proof that the interpretation of $\exists\alpha.\tau$ in Figure 5 is equivalence-respecting is given in our technical report [5].)

In Section 7, when proving Lemma 7.1, we make repeated use of the fact that all our type interpretations are equivalence-respecting.

Lemma 4.2 (Valid Type Interpretations)
If $\Delta \vdash \tau$ *and* $\rho \in \mathcal{D}\llbracket\Delta\rrbracket$, *then* $\mathcal{V}\llbracket\tau\rrbracket\,\rho \in \mathrm{Type}_{\rho_1(\tau),\rho_2(\tau)}$.

Computation relation. Two closed expressions e_1 and e_2 are related as computations of type τ for k steps (written $(k, e_1, e_2) \in \mathcal{C}\llbracket\tau\rrbracket\,\rho$) if the following holds. If e_1 evaluates to v_1 in $j < k$ steps, then e_2 must evaluate to some value v_2 (in any number of steps), and furthermore, the values v_1 and v_2 must be related for the remaining $k - j$ steps.

Asymmetric and symmetric relation. We say e_1 *logically approximates* e_2, written $\Delta; \Gamma \vdash e_1 \leq e_2 : \tau$ (see Figure 5), when e_1 and e_2 are both well-typed, and for all $k \geq 0$ steps, if ρ is a type substitution in $\mathcal{D}\llbracket\Delta\rrbracket$ and γ_1, γ_2 are mappings from variables x to closed values that are related for k steps at the types prescribed by Γ, then $\gamma_1(e_1)$ and $\gamma_2(e_2)$ are related for k steps as computations of type τ. We say e_1 and e_2 are *logically equivalent* (the symmetric relation, written $\Delta; \Gamma \vdash e_1 \approx e_2 : \tau$) if they logically approximate one another. The proofs of the following lemmas essentially follow those given in earlier work [3]; proofs of cases that involve existential types (as well as sum types, which were not considered in Ahmed's earlier work [4, 3]) are given in our technical report [5].

Lemma 4.3 (Fundamental Property)
If $\Delta; \Gamma \vdash e : \tau$ *then* $\Delta; \Gamma \vdash e \leq e : \tau$.

Lemma 4.4 (Sound & Complete w.r.t. Contextual Equivalence)
$\Delta; \Gamma \vdash e_1 \approx e_2 : \tau$ *iff* $\Delta; \Gamma \vdash e_1 \approx^{ctx} e_2 : \tau$.

5. Typed Closure Conversion

Closure conversion provides mappings from source values v of type τ to corresponding target values of type τ^+. We refer to τ^+ as the *translation type* corresponding to type τ. The rules for mapping τ to τ^+ (and Γ to Γ^+) are as follows.

$$\begin{array}{rcl}
\alpha^+ & = & \alpha \\
\mathrm{int}^+ & = & \mathrm{int} \\
(\exists\alpha.\tau)^+ & = & \exists\alpha.(\tau^+) \\
(\tau_1 \times \cdots \times \tau_k)^+ & = & \tau_1^+ \times \cdots \times \tau_k^+ \\
(\tau_1 + \tau_2)^+ & = & \tau_1^+ + \tau_2^+ \\
(\mu\alpha.\tau)^+ & = & \mu\alpha.(\tau^+) \\
(\forall[\bar{\alpha}](\bar{\tau})\to\sigma)^+ & = & \exists\eta.(\forall[\bar{\alpha}](\bar{\tau}^+, \eta)\to\sigma^+) \times \eta \\
 & & \\
(\cdot)^+ & = & \cdot \\
(\Gamma, x:\tau)^+ & = & \Gamma^+, x:\tau^+
\end{array}$$

Each function turns into an existential package that abstracts over the values of the function's free variables, i.e., its *closure environment*. The package itself contains a pair consisting of the *function pointer* and the closure environment. The closure environment is a tuple of values corresponding to the free variables. The function pointer is a closed function that takes the closure environment as an additional argument. The witness type of the package, abstractly represented by η, is the type of the closure environment. Notice that we use the same technique for closure conversion that was used in *System F to TAL* [21], which simplifies the approach taken by Minamide et al. [20] by closing only over free term variables, not over free type variables.

Figure 6 shows the rules for closure conversion in combination with declarative typing rules for the source language. To this end, the typing judgment $\Delta; \Gamma \vdash e : \tau$ is extended to a translation judgment $\Delta; \Gamma \vdash e : \tau \rightsquigarrow \hat{e}$ which states that the term e of type τ translates to the closure-converted term $\hat{e}$ that—under Γ^+—has type τ^+. Most of the rules are straightforward, the interesting ones being those for λ and application.

6. Wrappers

Given some type τ we want to construct function terms that map source values of type τ^- to translation values of type τ^+ and vice versa.[4] The construction proceeds in a syntax-directed way, using the structure of the type τ itself.

To illustrate the idea, consider the type $\tau = \mathrm{int} \to \mathrm{int}$. In this case, $\tau^+ = \exists\eta.((\mathrm{int}, \eta) \to \mathrm{int}) \times \eta$. To obtain a source value of type τ, we must unpack a closure of type τ^+, extract its components (a closed function and an environment), and yield a λ-term that takes an int argument x and applies the extracted function to both the argument x and the extracted environment. Similarly, we can turn a function f of type τ into a closure of translation type τ^+ by packing it with an environment that consists only of the original f. Thus, the two translation terms $\mathcal{W}^+_{\llbracket\tau\rrbracket}$ (function from τ to τ^+) and $\mathcal{W}^-_{\llbracket\tau\rrbracket}$ (function from τ^+ to τ) are:

$$\begin{array}{rl}
\mathcal{W}^+_{\llbracket\mathrm{int}\to\mathrm{int}\rrbracket} & = \lambda f : \mathrm{int}\to\mathrm{int}.\ \mathbf{pack}\ \mathrm{int}\to\mathrm{int}, (\lambda(x, f).\ f\,x, f) \\
 & \qquad\qquad\qquad\quad \mathbf{as}\ \exists\eta.((\mathrm{int}, \eta)\to\mathrm{int}) \times \eta \\
\mathcal{W}^-_{\llbracket\mathrm{int}\to\mathrm{int}\rrbracket} & = \lambda y : \exists\eta.((\mathrm{int}, \eta)\to\mathrm{int}) \times \eta. \\
 & \qquad \mathbf{unpack}\ [\zeta, z] = y\ \mathbf{in}\ \lambda x : \mathrm{int}.\ (\pi_1(z))\,(x, \pi_2(z))
\end{array}$$

Of course, if the co-domain of the function is more complicated than just int, then the function result must be further mapped appropriately. Similarly, if the domain is a structured type, then argument values must also be mapped—but, as usual in such situations, in the *opposite* direction: the polarity of the translation changes in contravariant positions.

[4] To be able to give a unified account of many of the symmetric constructions, we use the convention that τ^- is the same as τ.

$$\boxed{\Delta; \Gamma \vdash e : \tau \rightsquigarrow \hat{e}}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau \rightsquigarrow x}\ \textsc{var} \qquad \frac{}{\Delta; \Gamma \vdash c : \mathrm{int} \rightsquigarrow c}\ \textsc{const}$$

$$\frac{\Delta, \bar{\alpha}; \Gamma, \overline{x : \tau} \vdash e : \sigma \rightsquigarrow \hat{e} \qquad y_1, \ldots, y_m = \mathrm{FV}(\lambda[\bar{\alpha}](\overline{x : \tau}).\, e)}{\begin{array}{l}\Delta; \Gamma \vdash \lambda[\bar{\alpha}](\overline{x : \tau}).\, e : \forall[\bar{\alpha}](\bar{\tau}) \to \sigma \\ \quad \rightsquigarrow \quad \mathbf{pack}\,[\Gamma(y_1)^+ \times \cdots \times \Gamma(y_m)^+,\ (\lambda[\bar{\alpha}](\overline{x : \tau^+}, z : \Gamma(y_1)^+ \times \cdots \times \Gamma(y_m)^+). \\ \qquad\qquad \mathbf{let}\, y_1 = \pi_1(z)\,\mathbf{in} \ldots \mathbf{let}\, y_m = \pi_m(z)\,\mathbf{in}\, \hat{e}, \\ \qquad\qquad (y_1, \ldots, y_m))] \\ \qquad \mathbf{as}\, \exists \eta.(\forall[\bar{\alpha}](\bar{\tau}^+, \eta) \to \sigma^+) \times \eta\end{array}}\ \forall{\to}\text{-I}$$

$$\frac{\Delta; \Gamma \vdash e_0 : \forall[\bar{\alpha}](\tau_1, \ldots, \tau_n) \to \tau \rightsquigarrow \hat{e}_0 \qquad \overline{\Delta \vdash \sigma} \qquad \Delta; \Gamma \vdash e_1 : \tau_1[\overline{\sigma/\alpha}] \rightsquigarrow \hat{e}_1 \ \cdots \ \Delta; \Gamma \vdash e_n : \tau_n[\overline{\sigma/\alpha}] \rightsquigarrow \hat{e}_n}{\Delta; \Gamma \vdash e_0[\bar{\sigma}](e_1, \ldots, e_n) : \tau[\overline{\sigma/\alpha}] \rightsquigarrow \mathbf{unpack}\,[\zeta, z] = \hat{e}_0\,\mathbf{in}\,\pi_1(z)[\bar{\sigma}^+](\hat{e}_1, \ldots, \hat{e}_n, \pi_2(z))}\ \forall{\to}\text{-E}$$

$$\frac{\Delta; \Gamma \vdash e : \tau[\sigma/\alpha] \rightsquigarrow \hat{e} \qquad \Delta \vdash \sigma}{\Delta; \Gamma \vdash \mathbf{pack}\,[\sigma, e]\,\mathbf{as}\,\exists\alpha.\tau : \exists\alpha.\tau \rightsquigarrow \mathbf{pack}\,[\sigma^+, \hat{e}]\,\mathbf{as}\,\exists\alpha.\tau^+}\ \exists\text{-I} \qquad \frac{\Delta \vdash \tau' \qquad \Delta; \Gamma \vdash e_1 : \exists\alpha.\tau \rightsquigarrow \hat{e}_1 \qquad \Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau' \rightsquigarrow \hat{e}_2}{\Delta; \Gamma \vdash \mathbf{unpack}\,[\alpha, x] = e_1\,\mathbf{in}\,e_2 : \tau' \rightsquigarrow \mathbf{unpack}\,[\alpha, x] = \hat{e}_1\,\mathbf{in}\,\hat{e_2}}\ \exists\text{-E}$$

$$\frac{\Delta; \Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha] \rightsquigarrow \hat{e}}{\Delta; \Gamma \vdash \mathbf{fold}[\mu\alpha.\tau]e : \mu\alpha.\tau \rightsquigarrow \mathbf{fold}[\mu\alpha.\tau^+]\hat{e}}\ \mu\text{-I} \qquad \frac{\Delta; \Gamma \vdash e : \mu\alpha.\tau \rightsquigarrow \hat{e}}{\Delta; \Gamma \vdash \mathbf{unfold}\, e : \tau[(\mu\alpha.\tau)/\alpha] \rightsquigarrow \mathbf{unfold}\, \hat{e}}\ \mu\text{-E}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \rightsquigarrow \hat{e}_1 \ \cdots \ \Delta; \Gamma \vdash e_n : \tau_n \rightsquigarrow \hat{e}_n}{\Delta; \Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \cdots \times \tau_n \rightsquigarrow (\hat{e}_1, \ldots, \hat{e}_n)}\ \times\text{-I} \qquad \frac{\Delta; \Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \rightsquigarrow \hat{e}}{\Delta; \Gamma \vdash \pi_i(e) : \tau_i \rightsquigarrow \pi_i(\hat{e})}\ \times\text{-E} \qquad \frac{\Delta; \Gamma \vdash e : \tau_1 \rightsquigarrow \hat{e} \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash \mathbf{inl}(e) : \tau_1 + \tau_2 \rightsquigarrow \mathbf{inl}(\hat{e})}\ +_l\text{-I}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_2 \rightsquigarrow \hat{e} \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \mathbf{inr}(e) : \tau_1 + \tau_2 \rightsquigarrow \mathbf{inr}(\hat{e})}\ +_r\text{-I} \qquad \frac{\Delta; \Gamma \vdash e_0 : \tau_1 + \tau_2 \rightsquigarrow \hat{e}_0 \qquad \Delta; \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \rightsquigarrow \hat{e}_1 \qquad \Delta; \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \rightsquigarrow \hat{e}_2}{\begin{array}{c}\Delta; \Gamma \vdash \mathbf{case}\, e_0\, \mathbf{of}\, \mathbf{inl}(x_1) \Rightarrow e_1 \mid \mathbf{inr}(x_2) \Rightarrow e_2 : \tau \\ \rightsquigarrow \mathbf{case}\, \hat{e}_0\, \mathbf{of}\, \mathbf{inl}(x_1) \Rightarrow \hat{e}_1 \mid \mathbf{inr}(x_2) \Rightarrow \hat{e}_2\end{array}}\ +\text{-E}$$

Figure 6. Closure conversion

$$\begin{array}{rcl}
\mathcal{W}^{\pm}_{[\![\alpha;R]\!]} & = & \left\{\begin{array}{ll} R^{\pm}(\alpha) & ;\text{if } \alpha \in \mathrm{dom}(R) \\ \lambda x : \alpha.\, x & ;\text{if } \alpha \notin \mathrm{dom}(R) \end{array}\right. \\
\mathcal{W}^{\pm}_{[\![\mathrm{int};R]\!]} & = & \lambda x : \mathrm{int}.\, x \\
\mathcal{W}^{\pm}_{[\![\exists\alpha.\tau;R]\!]} & = & \lambda x : \exists\alpha.\tau^{\mp}.\, \mathbf{unpack}\,[\alpha, y] = x\,\mathbf{in}\,\mathbf{pack}\,[\alpha, \mathcal{W}^{\pm}_{[\![\tau;R]\!]}(y)]\,\mathbf{as}\,\exists\alpha.\tau^{\pm} \\
\mathcal{W}^{\pm}_{[\![\tau_1 \times \cdots \times \tau_k;R]\!]} & = & \lambda x : \tau_1^{\mp} \times \cdots \times \tau_k^{\mp}.\, (\mathcal{W}^{\pm}_{[\![\tau_1;R]\!]}(\pi_1(x)), \ldots, \mathcal{W}^{\pm}_{[\![\tau_k;R]\!]}(\pi_k(x))) \\
\mathcal{W}^{\pm}_{[\![\tau_1 + \tau_2;R]\!]} & = & \lambda x : \tau_1^{\mp} + \tau_2^{\mp}.\, \mathbf{case}\, x\, \mathbf{of}\, \mathbf{inl}(x_1) \Rightarrow \mathbf{inl}(\mathcal{W}^{\pm}_{[\![\tau_1;R]\!]}(x_1)) \mid \mathbf{inr}(x_2) \Rightarrow \mathbf{inr}(\mathcal{W}^{\pm}_{[\![\tau_2;R]\!]}(x_2)) \\
\mathcal{W}^{\pm}_{[\![\mu\alpha.\tau;R]\!]} & = & \mathbf{letrec}\, r^- : \mu\alpha.\tau^+ \to \mu\alpha.\tau = \lambda x.\, \mathbf{fold}[\mu\alpha.\tau](\mathcal{W}^{-}_{[\![\tau;R[\alpha \mapsto (r^-, r^+)]]\!]}(\mathbf{unfold}\, x)) \\
 & & \mathbf{and} \quad r^+ : \mu\alpha.\tau \to \mu\alpha.\tau^+ = \lambda x.\, \mathbf{fold}[\mu\alpha.\tau^+](\mathcal{W}^{+}_{[\![\tau;R[\alpha \mapsto (r^-, r^+)]]\!]}(\mathbf{unfold}\, x)) \\
 & & \mathbf{in}\ r^{\pm} \\
\mathcal{W}^{-}_{[\![\forall[\bar{\alpha}](\bar{\tau}) \to \sigma;R]\!]} & = & \lambda x : \exists\eta.(\forall[\bar{\alpha}](\overline{\tau^+}, \eta) \to \sigma^+) \times \eta.\, \mathbf{unpack}\,[\zeta, z] = x\,\mathbf{in}\,\lambda[\bar{\alpha}](\overline{x : \tau}).\, \mathcal{W}^{-}_{[\![\sigma;R]\!]}(\pi_1(z)[\bar{\alpha}](\overline{\mathcal{W}^{+}_{[\![\tau;R]\!]}(x)}, \pi_2(z))) \\
\mathcal{W}^{+}_{[\![\forall[\bar{\alpha}](\bar{\tau}) \to \sigma;R]\!]} & = & \lambda f : \forall[\bar{\alpha}](\bar{\tau}) \to \sigma.\, \mathbf{pack}\,[\forall[\bar{\alpha}](\bar{\tau}) \to \sigma, (\lambda[\bar{\alpha}](\overline{x : \tau^+}, f : \forall[\bar{\alpha}](\bar{\tau}) \to \sigma).\, \mathcal{W}^{+}_{[\![\sigma;R]\!]}(f[\bar{\alpha}](\overline{\mathcal{W}^{-}_{[\![\tau;R]\!]}(x))}), f)] \\
 & & \qquad\qquad \mathbf{as}\, \exists\eta.(\forall[\bar{\alpha}](\bar{\tau}^+, \eta) \to \sigma^+) \times \eta
\end{array}$$

Figure 7. Type-directed wrapper construction

The overall construction is shown in Figure 7. There are two difficulties, each having to do with type variables and their binding sites: abstract types and recursive types.

Abstract types. Translation works directly on the syntax of the source term. Therefore, it can—and does—manipulate the internals of abstractions, including the bodies of λ- or **pack**-expressions. Wrappers, on the other hand, have no access to the internal term structure of their arguments. They themselves are just terms within the language and have to respect abstractions. This seemingly serious limitation manifests itself as follows: type variables bound by $\forall$ or $\exists$ represent abstract types. When our type-directed wrapper construction encounters an abstract type α, it has no way of knowing anything about the values inhabiting α. Therefore, it has no choice but to leave such values unchanged. In other words, wrappers can never go "under an abstraction." Or, using the terminology of Section 2, wrappers are unable to "back-translate" values that are abstract. Fortunately, the problem is its own solution here: no context can observe this failure to back-translate since, after all, these values are abstract!

Recursive types. Type variables bound by μ are not abstract; they represent recursive types. When constructing a wrapper for $\mu\alpha.\tau$ it is tempting to unfold and rely on the corresponding wrapper for $\tau[\mu\alpha.\tau/\alpha]$. While in principle this seems like the right idea, it does not work because the construction would not be well-founded. The solution is to defer the unfolding step until the time when the wrapper is actually applied. We do this by mirroring the recursive type structure using a recursive term structure: we define the wrappers for $\mu\alpha.\tau$ using a pair of $\mathbf{letrec}$-bound recursive functions r^- and r^+. Within the body τ of the recursive type, any occurrence of α is to be treated not as an abstract type but as the same recursive type for which we have r^- and r^+ in place. For this, the construction of the wrapper for τ carries a mapping R that associates α with the pair (r^-, r^+) of function names.

Formally, suppose $\Delta \vdash \tau$. Let Δ be partitioned into disjoint sets of variables Δ_a (for abstract types) and Δ_r (for recursive types), and let R be a mapping from the variables in Δ_r to pairs of (possibly open) terms (r^-, r^+) which we will refer to as $R^-(\alpha)$ and $R^+(\alpha)$, respectively. Then $\mathcal{W}^+_{\llbracket\tau;R\rrbracket}$ denotes the forward-wrapper term constructed for τ and R, while $\mathcal{W}^-_{\llbracket\tau;R\rrbracket}$ denotes the backward-wrapper term.

The only place where we need to concern ourselves with types that have open recursive type variables is when we construct wrappers and—to make inductions go through—when we prove certain facts about them. Once the machinery is in place (i.e., once the required lemmas have been proved), we will consider only cases where $\Delta_r = \mathrm{dom}(R) = \emptyset$. For this, we write $\mathcal{W}^\pm_{\llbracket\tau\rrbracket}$ to mean $\mathcal{W}^\pm_{\llbracket\tau;\emptyset\rrbracket}$.

6.1 Wrapper contexts

Let $\Delta; \Gamma \vdash e : \tau \rightsquigarrow \hat{e}$. In the general case, the translation term $\hat{e}$ is not well-typed under the source-level environment Γ. To make it well-typed, we can consider it in a context that $\mathbf{let}$-binds each of its free variables to a suitable translation. The notion of *wrapper contexts* makes this idea precise. The context $\mathcal{W}^\pm_{\llbracket\Gamma\rrbracket}$ consists of a sequence of $\mathbf{let}$-bindings of the form $\mathbf{let}\, x = \mathcal{W}^\pm_{\llbracket\tau\rrbracket}(x)\, \mathbf{in} \cdots$, one for each $x : \tau$ in Γ. Formally:

$$\begin{aligned} \mathcal{W}^\pm_{\llbracket\cdot\rrbracket} &= [\cdot] \\ \mathcal{W}^\pm_{\llbracket\Gamma, x:\tau\rrbracket} &= \mathbf{let}\, x = \mathcal{W}^\pm_{\llbracket\tau\rrbracket}(x)\, \mathbf{in}\, \mathcal{W}^\pm_{\llbracket\Gamma\rrbracket} \end{aligned}$$

6.2 Properties of wrappers

We will now establish a number of facts about wrappers. These facts will play a crucial role in our investigation of how wrappers relate to closure conversion. We start with the typing of wrappers, then discuss wrapper termination (the fact that wrapper functions are total), and wrapper cancellation (the fact that pairs $(\mathcal{W}^+_{\llbracket\tau\rrbracket}, \mathcal{W}^-_{\llbracket\tau\rrbracket})$ establish an isomorphism between the equivalence classes of τ and τ^+). Then, to make it easier to work with wrappers at recursive types, we state a *wrapper unfolding* lemma. Finally, we will discuss *wrapper parametricity*, which is *the* essential property that justifies the behavior of wrappers at abstract types.

Typing. Let $\Delta = \Delta_a \uplus \Delta_r$ and $\mathrm{dom}(R) = \Delta_r$. Let also $\delta_r \vDash \Delta_r$ (meaning δ_r is a type assignment to variables in Δ_r) and a typing environment Γ_r such that $\Delta_a; \Gamma_r \vdash R^\pm(\alpha) : \delta_r(\alpha)^\mp \to \delta_r(\alpha)^\pm$. Constructed wrappers then satisfy the following typing judgment:

$$\Delta_a; \Gamma_r \vdash \mathcal{W}^\pm_{\llbracket\tau;R\rrbracket} : \delta_r(\tau)^\mp \to \delta_r(\tau)^\pm$$

For types without open recursive variables this gives:

$$\Delta; \cdot \vdash \mathcal{W}^\pm_{\llbracket\tau\rrbracket} : \tau^\mp \to \tau^\pm$$

Termination. We show that wrapper functions are total:

Lemma 6.1 (wrapper termination)
Let $\Delta \vdash \tau$ and $\delta \vDash \Delta$. If $\vdash v : \delta(\tau^\pm)$, then there is some finite j and some v' of type $\delta(\tau^\mp)$ such that $(\delta(\mathcal{W}^\mp_{\llbracket\tau\rrbracket}))\, v \longrightarrow^j v'$.

Proof: By induction on the structure of v. □

A consequence of wrapper termination is that wrapper contexts may be freely duplicated and, therefore, *distributed*, for example as in $\mathcal{W}^+_{\llbracket\Gamma\rrbracket}[e_1\, e_2] \approx (\mathcal{W}^+_{\llbracket\Gamma\rrbracket}[e_1])\, (\mathcal{W}^+_{\llbracket\Gamma\rrbracket}[e_2])$. Similarly, one can drop unneeded bindings from wrapper contexts. In particular, $\mathcal{W}^+_{\llbracket\Gamma\rrbracket}[x] \approx \mathcal{W}^+_{\llbracket\Gamma(x)\rrbracket}(x)$.

Cancellation. $\mathcal{W}^-_{\llbracket\tau\rrbracket}$ and $\mathcal{W}^+_{\llbracket\tau\rrbracket}$ are inverses of each other. They, therefore, mediate an isomorphism between (the equivalence classes of) τ and τ^+.

Because of recursion we have to state and prove this property in terms of our step-indexed logical relation. We start with the definition of $f^- \bowtie^\tau_k f^+$. Intuitively, it states that the cancellation property holds for (f^-, f^+) at source type τ for at least k steps:

Definition 6.2 (cancellation)
Let $\Delta \vdash \tau$ and $\Delta; \cdot \vdash f^\pm : \tau^\mp \to \tau^\pm$.

$$\begin{aligned} f^- \bowtie^\tau_k f^+ \quad &=_{\mathrm{def}} \quad \forall \rho \in \mathcal{D}\llbracket\Delta\rrbracket . \\ &\quad (k, v_1, v_2) \in \mathcal{V}\llbracket\tau^\pm\rrbracket\, \rho \Rightarrow (k, v_1, \rho_2(f^\pm(f^\mp\, v_2))) \in \mathcal{C}\llbracket\tau^\pm\rrbracket\, \rho \\ f^- \bowtie^\tau f^+ \quad &=_{\mathrm{def}} \quad \forall k.\, f^- \bowtie^\tau_k f^+ \end{aligned}$$

If two values cancel for any number of steps, then they are inverses of each other: if $v_1 \bowtie^\tau v_2$, then $\Delta; \cdot \vdash v_1 \circ v_2 \approx \mathsf{id} : \tau \to \tau$ and $\Delta; \cdot \vdash v_2 \circ v_1 \approx \mathsf{id} : \tau^+ \to \tau^+$.

To deal with the distinction between abstract and recursive type variables, we define the notion of a type partition:

Definition 6.3 (type partition)
Let Δ be a set of type variables. We write $\Delta \rightsquigarrow^{\bowtie}_k (\Delta_a, \Delta_r, \delta_r, R, \gamma_r)$ if there exists a Γ_a such that the following holds:

$$\begin{aligned} &\Delta = \Delta_a \uplus \Delta_r \ \wedge\ \mathrm{dom}(R) = \Delta_r \ \wedge \\ &\forall \alpha \in \Delta_r.\ \ \Delta_a; \Gamma_r \vdash R^\pm(\alpha) : \delta_r(\alpha)^\mp \to \delta_r(\alpha)^\pm \ \wedge \\ &\qquad\qquad \gamma_r(R^-(\alpha)) \bowtie^{\delta_r(\alpha)}_k \gamma_r(R^+(\alpha)) \end{aligned}$$

This definition should be read as follows: Δ is partitioned into sets of abstract variables Δ_a and recursive variables Δ_r. The type substitution δ_r maps each $\alpha \in \Delta_r$ to its corresponding type,[5] which itself must be well-formed in Δ_a. The mapping R assigns two terms to each recursive type variable: the *backward*-wrapper $R(\alpha)^-$ and the *forward*-wrapper $R(\alpha)^+$. Since these terms may contain free variables, we need a typing environment Γ_r relative to which they have the correct types. Finally, we have a closing substitution γ_r that respects Γ_r and turns each of the wrapper pairs in R into a canceling pair.

Given a type τ that is well-formed in Δ together with a type partition of Δ, the corresponding wrapper pair cancels:

Lemma 6.4
Let $\Delta \vdash \tau$. For any $k \geq 0$, if $\Delta \rightsquigarrow^{\bowtie}_k (\Delta_a, \Delta_r, \delta_r, R, \gamma_r)$, then $\gamma_r(\mathcal{W}^-_{\llbracket\tau;R\rrbracket}) \bowtie^{\delta_r(\tau)}_k \gamma_r(\mathcal{W}^+_{\llbracket\tau;R\rrbracket})$.

Proof: By outer induction on k and inner induction on τ. □

For types without open recursive variables this leads to the following much simpler statement:

Lemma 6.5 (wrapper cancellation)
Let $\Delta; \Gamma \vdash e : \tau^\pm$. Then $\Delta; \Gamma \vdash e \approx \mathcal{W}^\pm_{\llbracket\tau\rrbracket}(\mathcal{W}^\mp_{\llbracket\tau\rrbracket}(e)) : \tau^\pm$.

[5] This type will always turn out to be of the form $\mu\alpha.\tau$ for some τ.

$$\begin{array}{rcll}
\mathcal{W}^{-[+]}_{\{\tau,\sigma/\alpha\}} & = & (\mathcal{W}^{-}_{\llbracket\tau\rrbracket})[\sigma/\alpha] \circ \mathcal{W}^{+}_{\llbracket\tau[\sigma/\alpha]\rrbracket} & : \tau[\sigma/\alpha] \to \tau[\sigma^{+}/\alpha] \\
\mathcal{W}^{-[-]}_{\{\tau,\sigma/\alpha\}} & = & \mathcal{W}^{-}_{\llbracket\tau[\sigma/\alpha]\rrbracket} \circ (\mathcal{W}^{+}_{\llbracket\tau\rrbracket})[\sigma^{+}/\alpha] & : \tau[\sigma^{+}/\alpha] \to \tau[\sigma/\alpha] \\
\mathcal{W}^{+[+]}_{\{\tau^{+},\sigma/\alpha\}} & = & \mathcal{W}^{+}_{\llbracket\tau[\sigma/\alpha]\rrbracket} \circ (\mathcal{W}^{-}_{\llbracket\tau\rrbracket}[\sigma/\alpha]) & : \tau^{+}[\sigma/\alpha] \to \tau^{+}[\sigma^{+}/\alpha] \\
\mathcal{W}^{+[-]}_{\{\tau^{+},\sigma/\alpha\}} & = & (\mathcal{W}^{+}_{\llbracket\tau\rrbracket})[\sigma/\alpha] \circ \mathcal{W}^{-}_{\llbracket\tau[\sigma/\alpha]\rrbracket} & : \tau^{+}[\sigma^{+}/\alpha] \to \tau^{+}[\sigma/\alpha]
\end{array}$$

Figure 8. Combination wrappers

Proof: Using Lemma 6.4 and the trivial partition $\Delta \rightsquigarrow^{\bowtie}_{k} (\Delta, \emptyset, \emptyset, \emptyset, \emptyset)$. □

Unfolding. As we have explained, the point of the $\mathbf{letrec}$-construction is for $\mathcal{W}^{+}_{\llbracket\mu\alpha.\tau\rrbracket}$ to be a *finite* term. However, that term still should relate to $\mathcal{W}^{+}_{\llbracket\tau[\mu\alpha.\tau/\alpha]\rrbracket}$ *as if* it had been constructed using the unfolding of $\mu\alpha.\tau$. The property in question is formally expressed in Lemma 6.7 below.

To prove it, we must again strengthen the induction hypothesis and make use of step-indexing. This leads to the following lemma:[6]

Lemma 6.6
Let $\Delta \vdash \mu\alpha.\tau$, *and* $\tau_u = \tau[\mu\alpha.\tau/\alpha]$. *For any* $k \geq 0$, *if* $\Delta \rightsquigarrow^{\bowtie}_{k} (\Delta_a, \Delta_r, \delta_r, R, \gamma_r)$, *then*

$$\begin{array}{l}
\forall\rho \in \mathcal{D}\llbracket\Delta_a\rrbracket\,. \\
\quad \forall(k, v_1, v_2) \in \mathcal{V}\left\llbracket\delta_r(\tau_u^{+})\right\rrbracket \rho. \\
\qquad (k, \rho_1(\gamma_r(\mathcal{W}^{-}_{\llbracket\tau_u;R\rrbracket}(v_1))), \rho_2(\gamma_r(\mathcal{W}^{-}_{\llbracket\tau;R'\rrbracket}(v_2)))) \\
\qquad \in \mathcal{C}\llbracket\delta_r(\tau_u)\rrbracket \rho \\
\quad \textit{where } R' = R[\alpha \mapsto (\mathcal{W}^{-}_{\llbracket\tau_u;R\rrbracket}, \mathcal{W}^{+}_{\llbracket\tau_u;R\rrbracket})]
\end{array}$$

Proof: By outer induction on k and inner induction on τ. □

Lemma 6.7 (wrapper unfolding)
Let $\tau_u = \tau[\mu\alpha.\tau/\alpha]$. *If* $\Delta; \Gamma \vdash e : \tau_u^{+}$, *then*

$$\begin{array}{l}
\Delta; \Gamma \vdash \mathcal{W}^{-}_{\llbracket\mu\alpha.\tau\rrbracket}(\mathbf{fold}[\mu\alpha.\tau^{+}]e) \\
\qquad\qquad \approx \mathbf{fold}[\mu\alpha.\tau](\mathcal{W}^{-}_{\llbracket\tau_u\rrbracket}(e)) : \mu\alpha.\tau.
\end{array}$$

Proof sketch: We use the observation, implied by Lemma 6.6, that $\mathcal{W}^{-}_{\llbracket\tau_u;R\rrbracket} \approx \mathcal{W}^{-}_{\left\llbracket\tau;R[\alpha\mapsto(\mathcal{W}^{-}_{\llbracket\tau_u;R\rrbracket},\mathcal{W}^{+}_{\llbracket\tau_u;R\rrbracket})]\right\rrbracket}$. □

Lemma 6.7 can be stated in several other ways, all of which are inter-derivable by equivalent transformations:

$$\begin{array}{rcl}
\mathbf{fold}[\mu\alpha.\tau^{+}](\mathcal{W}^{+}_{\llbracket\tau_u\rrbracket}(e)) & \approx & \mathcal{W}^{+}_{\llbracket\mu\alpha.\tau\rrbracket}(\mathbf{fold}[\mu\alpha.\tau]e) \\
\mathbf{unfold}\,(\mathcal{W}^{-}_{\llbracket\mu\alpha.\tau\rrbracket}(e)) & \approx & \mathcal{W}^{-}_{\llbracket\tau_u\rrbracket}(\mathbf{unfold}\,e) \\
\mathcal{W}^{+}_{\llbracket\tau_u\rrbracket}(\mathbf{unfold}\,e) & \approx & \mathbf{unfold}\,(\mathcal{W}^{+}_{\llbracket\mu\alpha.\tau\rrbracket}(e))
\end{array}$$

Parametricity. Consider the instantiation $\tau[\sigma/\alpha]$ of some type τ with a free type variable α. Consider two wrappers, one for the fully instantiated type ($\mathcal{W}^{+}_{\llbracket\tau[\sigma/\alpha]\rrbracket}$) and the other for unwrapping τ while holding α abstract ($(\mathcal{W}^{-}_{\llbracket\tau\rrbracket})[\sigma/\alpha]$). Their composition $(\mathcal{W}^{-}_{\llbracket\tau\rrbracket})[\sigma/\alpha] \circ \mathcal{W}^{+}_{\llbracket\tau[\sigma/\alpha]\rrbracket}$ is a wrapper for σ that acts just at the occurrences of α in τ. Let us use the notation $\mathcal{W}^{-[+]}_{\{\tau,\sigma/\alpha\}}$ for this combination. Other possible combinations are shown is Figure 8.

The logical relation gives rise to several inter-derivable *free theorems* stating that combination wrappers can be "pushed through" applications of polymorphic functions. For example, given a function $f : \forall[\alpha](\tau) \to \tau_r$ we have:

$$\mathcal{W}^{-[+]}_{\{\tau_r,\sigma/\alpha\}}(f[\sigma]v) \approx f[\sigma^{+}](\mathcal{W}^{-[+]}_{\{\tau,\sigma/\alpha\}}(v))$$

The following lemma states this more generally (but still, *w.l.o.g.*, only for the first type argument):

Lemma 6.8 (wrapper parametricity)
Let $\Delta; \Gamma \vdash e_0 : \forall[\alpha_1, \bar{\alpha}](\bar{\tau}) \to \tau_r$ *and* $\bar{e} = e_1, \ldots, e_n$ *such that* $\Delta; \Gamma \vdash e_i : \tau_i[\sigma_1/\alpha_1, \overline{\sigma/\alpha}]$ *for* $1 \leq i \leq n$. *Then*

$$\begin{array}{l}
\Delta; \Gamma \vdash e_0[\sigma_1^{\pm}, \bar{\sigma}]\overline{(\mathcal{W}^{-[\pm]}_{\{\tau[\overline{\sigma/\alpha}],\sigma_1/\alpha_1\}}(e))} \approx \\
\qquad\qquad \mathcal{W}^{-[\pm]}_{\{\tau_r[\overline{\sigma/\alpha}],\sigma/\alpha\}}(e_0[\sigma_1^{\mp}, \bar{\sigma}](\bar{e})) : \tau_r[\sigma_1^{\pm}/\alpha_1, \overline{\sigma/\alpha}]
\end{array}$$

An analogous statement holds for $\mathcal{W}^{+[\pm]}$.

Proof sketch: By instantiating the definition of the logical relation for the type of e_0, mapping α_1 to $(\sigma_1^{\pm}, \sigma_1^{\mp}, \chi^{*\mathrm{ciu}}_{\sigma_1^{\mp}})$ such that $(i, v, v') \in \chi$ iff $\mathcal{W}^{\mp}_{\llbracket\sigma_1\rrbracket}v \longrightarrow^{*} v'$. □
By the same argument, the following corollary concerning existentials holds as well:

Lemma 6.9 (wrapper abstraction)
If $\Delta; \Gamma \vdash e : \tau[\sigma/\alpha]^{\pm}$, *then*

$$\begin{array}{l}
\Delta; \Gamma \vdash \mathcal{W}^{\mp}_{\llbracket\exists\alpha.\tau\rrbracket}(\mathbf{pack}\,[\sigma^{\pm}, e]\,\mathbf{as}\,\exists\alpha.\tau^{\pm}) \approx \\
\qquad\qquad \mathbf{pack}\,[\sigma^{\mp}, \mathcal{W}^{\mp}_{\llbracket\tau[\sigma/\alpha]\rrbracket}(e)]\,\mathbf{as}\,\exists\alpha.\tau^{\mp} : \exists\alpha.\tau^{\mp}
\end{array}$$

7. Translation is Equivalent to Wrapping

The main result of this paper, namely that typed closure conversion is equivalence-preserving, is a consequence of the fact that the translation $\hat{e}$ of a term e is related (i.e., observationally equivalent) to the corresponding wrapped source term. For closed $e : \tau$ and $\hat{e} : \tau^{+}$, this fact an be written simply as: $e \approx \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\hat{e})$, or, equivalently, as $\hat{e} \approx \mathcal{W}^{+}_{\llbracket\tau\rrbracket}(e)$. Free variables, given by the typing environment Γ, can be accounted for by adding the appropriate context wrappers $\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}$ and $\mathcal{W}^{-}_{\llbracket\Gamma\rrbracket}$, giving rise to the following lemma:

Lemma 7.1 (Translation and wrapping are equivalent)
Let $\Delta; \Gamma \vdash e : \tau \rightsquigarrow \hat{e}$. *Then* $\Delta; \Gamma \vdash e \approx \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\hat{e})] : \tau$ *and* $\Delta; \Gamma^{+} \vdash \hat{e} \approx \mathcal{W}^{-}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{+}_{\llbracket\tau\rrbracket}(e)]$.

Proof:

The proof for Lemma 7.1 proceeds by induction on the step index and cases on e. In all cases we rely on those properties of wrappers discussed in Section 6.2 and perform equivalent transformations on the right-hand side until the induction hypothesis applies directly. Figure 9 shows the proof for the most interesting cases, in particular: constants, variables, λ and application (each, *w.l.o.g.* shown with only one type- and one term argument), and finally $\mathbf{pack}$ and $\mathbf{unpack}$. The proof for the remaining cases is given in the accompanying technical report [5].□

8. Fully Abstract Translation

Our main result, namely that closure conversion is fully abstract, follows directly from the above translation-wrapper equivalence lemma (Lemma 7.1) and the fact that the equivalence relation $\approx$ is sound and complete with respect to contextual equivalence (Lemma 4.4).

[6] It is convenient to reuse the $\rightsquigarrow^{\bowtie}_{k}$ notation here. This makes Lemma 6.6 slightly weaker than it could be—but still strong enough to prove Lemma 6.7.

Case $\boxed{c : \mathrm{int}}$ Consider right-hand side: $\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\mathrm{int}\rrbracket}(c)] \approx c$ □

Case $\boxed{x : \tau}$ Have $\Gamma(x) = \tau$. Right-hand side: $\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau\rrbracket}(x)] \approx \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\mathcal{W}^{+}_{\llbracket\Gamma(x)\rrbracket}(x)) \approx \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\mathcal{W}^{+}_{\llbracket\tau\rrbracket}(x)) \approx x$ □

Case $\boxed{e_0 = \lambda[\alpha](x : \tau_1).\, e : \tau_0}$ where $\tau_0 = \forall[\alpha](\tau_1) \to \tau$ and $\tau_0^{+} = \exists\eta.\forall[\alpha](\tau_1^{+}, \eta) \to \tau^{+} \times \eta$.
Let $\mathrm{FV}(e_0) = \{y_1, \ldots, y_m\}$ and $T = \Gamma(y_1) \times \cdots \times \Gamma(y_m)$. Have $\Delta, \alpha; \Gamma, x : \tau_1 \vdash e : \tau$ and
$\hat{e}_0 = \mathbf{pack}\,[T^{+}, (\lambda[\alpha](x : \tau_1^{+}, z : T^{+}).\, \mathbf{let}\, y_1 = \pi_1(z)\ \ldots\ y_m = \pi_m(z)\, \mathbf{in}\, \hat{e}, (y_1, \ldots, y_m))]\, \mathbf{as}\, \tau_0^{+}$.
Consider right-hand side:

$$\begin{aligned}
& \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau_0\rrbracket}(\hat{e}_0)] \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\zeta, w] = \hat{e}_0\, \mathbf{in}\, \lambda[\alpha](x : \tau_1).\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\pi_1(w)[\alpha](\mathcal{W}^{+}_{\llbracket\tau_1\rrbracket}(x), \pi_2(w)))] && \text{(Def. } \mathcal{W}^{-}_{\llbracket\tau_0\rrbracket}, \beta\text{-reduce)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\lambda[\alpha](x : \tau_1).\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}((\lambda[\alpha](x : \tau_1^{+}, z : T^{+}).\, \mathbf{let}\, y_1 = \pi_1(z)\ \ldots\ y_m = \pi_m(z)\, \mathbf{in}\, \hat{e})[\alpha](\mathcal{W}^{+}_{\llbracket\tau_1\rrbracket}(x), (y_1, \ldots, y_m)))] \\
& && \text{(unpack)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\lambda[\alpha](x : \tau_1).\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\mathbf{let}\, x = \mathcal{W}^{+}_{\llbracket\tau_1\rrbracket}(x)\, \mathbf{in}\, \hat{e})] && \text{(}\beta\text{-reduce, project)} \\
\approx\ & \lambda[\alpha](x : \tau_1).\, \underbrace{\mathcal{W}^{+}_{\llbracket\Gamma, x:\tau_1\rrbracket}[\hat{e}[\tau]\mathcal{W}^{-}_{\llbracket\tau\rrbracket}]}_{\approx e} && \text{(env. wrapper)} \\
\approx\ & \lambda[\alpha](x : \tau_1).\, e && \square\quad \text{(IH)}
\end{aligned}$$

Case $\boxed{e_0[\sigma]e_1 : \tau_r}$ Let $\tau_0 = \forall[\alpha](\tau_1) \to \tau, \tau_a = \tau_1[\sigma/\alpha], \tau_r = \tau[\sigma/\alpha]$. Have: $\Delta; \Gamma \vdash e_0 : \tau_0 \leadsto \hat{e}_0$ and $\Delta; \Gamma \vdash e_1 : \tau_a \leadsto \hat{e}_1$ as well as $\Delta; \Gamma \vdash e_0[\sigma]e_1 : \tau_r \leadsto \mathbf{unpack}\,[\zeta, z] = \hat{e_0}\, \mathbf{in}\, \pi_1(z)[\sigma^{+}](\hat{e}_1, \pi_2(z))$.
Consider right-hand side:

$$\begin{aligned}
& \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau_r\rrbracket}(\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \pi_1(z)[\sigma^{+}](\hat{e}_1, \pi_2(z)))] \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau_r\rrbracket}(\pi_1(z)[\sigma^{+}](\hat{e}_1, \pi_2(z)))] && \text{(rearrange)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}[\sigma/\alpha](\mathcal{W}^{+}_{\llbracket\tau\rrbracket}[\sigma/\alpha](\mathcal{W}^{-}_{\llbracket\tau_r\rrbracket}(\pi_1(z)[\sigma^{+}](\hat{e}_1, \pi_2(z)))))] && \text{(Lemma 6.5)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}[\sigma/\alpha](\mathcal{W}^{+[-]}_{\{\tau^{+}, \sigma/\alpha\}}(\pi_1(z)[\sigma^{+}](\hat{e}_1, \pi_2(z))))] && \text{(def. } \mathcal{W}^{+[-]}_{\{\tau^{+}, \sigma/\alpha\}}\text{)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}[\sigma/\alpha](\pi_1(z)[\sigma^{+}](\mathcal{W}^{+[-]}_{\{\tau_1^{+}, \sigma/\alpha\}}(\hat{e}_1), \pi_2(z)))] && \text{(Lemma 6.8)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}[\sigma/\alpha](\pi_1(z)[\sigma^{+}](\mathcal{W}^{+}_{\llbracket\tau_1\rrbracket}[\sigma/\alpha](\mathcal{W}^{-}_{\llbracket\tau_a\rrbracket}(\hat{e}_1)), \pi_2(z)))] && \text{(def. } \mathcal{W}^{+[-]}_{\{\tau_1^{+}, \sigma/\alpha\}}\text{)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[(\lambda[\alpha](x : \tau_1).\, \mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\pi_1(z)[\alpha](\mathcal{W}^{+}_{\llbracket\tau_1\rrbracket}(x), \pi_2(z))))[\sigma](\mathcal{W}^{-}_{\llbracket\tau_a\rrbracket}(\hat{e}_1))] && \text{(inv. } \beta\text{)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[(\mathbf{unpack}\,[\zeta, z] = \hat{e}_0\, \mathbf{in}\, \lambda[\alpha](x : \tau_1).\, \mathcal{W}^{-}_{\llbracket\tau\rrbracket}(\pi_1(z)[\alpha](\mathcal{W}^{+}_{\llbracket\tau_1\rrbracket}(x), \pi_2(z))))[\sigma](\mathcal{W}^{-}_{\llbracket\tau_a\rrbracket}(\hat{e}_1))] && \text{(rearrange)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau_0\rrbracket}(\hat{e}_0)[\sigma](\mathcal{W}^{-}_{\llbracket\tau_a\rrbracket}(\hat{e}_1))] && \text{(def. } \mathcal{W}^{-}_{\llbracket\tau_0\rrbracket}\text{)} \\
\approx\ & \underbrace{(\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\hat{e}_0[\tau_0]\mathcal{W}^{-}_{\llbracket\tau_0\rrbracket}])}_{\approx e_0}[\sigma]\underbrace{(\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\hat{e}_1[\tau_a]\mathcal{W}^{-}_{\llbracket\tau_a\rrbracket}])}_{\approx e_1} && \text{(distrib. } \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}\text{)} \\
\approx\ & e_0[\sigma]e_1 && \square\quad \text{(IH)}
\end{aligned}$$

Case $\boxed{\mathbf{pack}\,[\sigma, e]\, \mathbf{as}\, \exists\alpha.\tau}$ where $e : \tau_p$ and $\tau_p = \tau[\sigma/\alpha]$:

$$\begin{aligned}
& \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\exists\alpha.\tau\rrbracket}(\mathbf{pack}\,[\sigma^{+}, \hat{e}]\, \mathbf{as}\, \exists\alpha.\tau^{+})] \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{pack}\,[\sigma, \mathcal{W}^{-}_{\llbracket\tau_p\rrbracket}(\hat{e})]\, \mathbf{as}\, \exists\alpha.\tau] && \text{(Lemma 6.9)} \\
\approx\ & \mathbf{pack}\,[\sigma, \underbrace{\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau_p\rrbracket}(\hat{e})]}_{\approx e}]\, \mathbf{as}\, \exists\alpha.\tau && \text{(distrib. } \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}\text{)} \\
\approx\ & \mathbf{pack}\,[\sigma, e]\, \mathbf{as}\, \exists\alpha.\tau && \square\quad \text{(IH)}
\end{aligned}$$

Case $\boxed{\mathbf{unpack}\,[\alpha, x] = e_1\, \mathbf{in}\, e_2 : \tau_2}$, where $\Delta; \Gamma \vdash e_1 : \exists\alpha.\tau$, and $\Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau_2$:

$$\begin{aligned}
& \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\mathbf{unpack}\,[\alpha, x] = \hat{e}_1\, \mathbf{in}\, \hat{e}_2)] \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\alpha, x] = \hat{e}_1\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\hat{e}_2)] && \text{(rearrange)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\alpha, x] = \mathcal{W}^{+}_{\llbracket\exists\alpha.\tau\rrbracket}(\mathcal{W}^{-}_{\llbracket\exists\alpha.\tau\rrbracket}(\hat{e}_1))\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\hat{e}_2)] && \text{(Lemma 6.5)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\alpha, x] = (\mathbf{unpack}\,[\alpha, x] = \mathcal{W}^{-}_{\llbracket\exists\alpha.\tau\rrbracket}(\hat{e}_1)\, \mathbf{in}\, \mathbf{pack}\,[\alpha, \mathcal{W}^{+}_{\llbracket\exists\alpha.\tau\rrbracket}(x)]\, \mathbf{as}\, \exists\alpha.\tau^{+})\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\hat{e}_2)] \\
& && \text{(def. } \mathcal{W}^{+}_{\llbracket\exists\alpha.\tau\rrbracket}\text{)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\alpha, x] = \mathcal{W}^{-}_{\llbracket\exists\alpha.\tau\rrbracket}(\hat{e}_1)\, \mathbf{in}\, \mathbf{unpack}\,[\alpha, x] = \mathbf{pack}\,[\alpha, \mathcal{W}^{+}_{\llbracket\tau\rrbracket}(x)]\, \mathbf{as}\, \exists\alpha.\tau^{+}\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\hat{e}_2)] \\
& && \text{(rearrange)} \\
\approx\ & \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathbf{unpack}\,[\alpha, x] = \mathcal{W}^{-}_{\llbracket\exists\alpha.\tau\rrbracket}(\hat{e}_1)\, \mathbf{in}\, \mathbf{let}\, x = \mathcal{W}^{+}_{\llbracket\tau\rrbracket}(x)\, \mathbf{in}\, \mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\hat{e}_2)] && \text{(unpack)} \\
\approx\ & \mathbf{unpack}\,[\alpha, x] = \underbrace{\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\exists\alpha.\tau\rrbracket}(\hat{e}_1)]}_{\approx e_1}\, \mathbf{in}\, \underbrace{\mathcal{W}^{+}_{\llbracket\Gamma, x:\tau\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau_2\rrbracket}(\hat{e}_2)]}_{\approx e_2} && \text{(distrib. } \mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}\text{)} \\
\approx\ & \mathbf{unpack}\,[\alpha, x] = e_1\, \mathbf{in}\, e_2 && \square\quad \text{(IH)}
\end{aligned}$$

Figure 9. Selected proof cases for Lemma 7.1.

$$\begin{aligned}
\mathcal{T}^{\xi}_{\mathrm{V}}\,\llbracket \tau_1 \to \tau_2 \rrbracket\,\rho \quad &= \quad \{\, (k, \lambda x : \tau_1 .\, e_1, \mathbf{pack}\,[\sigma, ((\lambda(x : \tau_1^{+}, z : \sigma).\, e_2), v)]\;\mathbf{as}\;\exists\eta.((\tau_1^{+}, \eta) \to \tau_2^{+}) \times \eta) \mid \\
&\qquad\qquad \forall j < k.\, \forall v_1, v_2.\ \text{if}\ (j, v_1, v_2) \in \mathcal{T}^{\xi}_{\mathrm{V}}\,\llbracket \tau_1 \rrbracket\,\rho\ \text{then}\ (j, e_1[v_1/x], e_2[v_2/x][v/z]) \in \mathcal{T}^{\xi}_{\mathrm{C}}\,\llbracket \tau_2 \rrbracket\,\rho \,\} \\
\mathcal{T}^{\xi}_{\mathrm{V}}\,\llbracket \ldots \rrbracket\,\rho \quad &= \quad \ldots \\
\mathcal{T}^{\leq}_{\mathrm{C}}\,\llbracket \tau \rrbracket\,\rho \quad &= \quad \{\, (k, e_1, e_2) \mid \forall j < k.\, \forall v_1.\ \text{if}\ e_1 \longrightarrow^{j} v_1\ \text{then}\ \exists v_2.\, e_2 \longrightarrow^{*} v_2 \ \wedge\ (k - j, v_1, v_2) \in \mathcal{T}^{\leq}_{\mathrm{V}}\,\llbracket \tau \rrbracket\,\rho \,\} \\
\mathcal{T}^{\geq}_{\mathrm{C}}\,\llbracket \tau \rrbracket\,\rho \quad &= \quad \{\, (k, e_1, e_2) \mid \forall j < k.\, \forall v_2.\ \text{if}\ e_2 \longrightarrow^{j} v_2\ \text{then}\ \exists v_1.\, e_1 \longrightarrow^{*} v_1 \ \wedge\ (k - j, v_1, v_2) \in \mathcal{T}^{\geq}_{\mathrm{V}}\,\llbracket \tau \rrbracket\,\rho \,\} \\
&\ldots \\
\Delta; \Gamma \vdash e_1 \propto e_2 : \tau \quad &=_{\mathrm{def}} \quad \Delta; \Gamma \vdash e_1 \propto_{\leq} e_2 : \tau \ \wedge\ \Delta; \Gamma \vdash e_1 \propto_{\geq} e_2 : \tau
\end{aligned}$$

Figure 10. Step-indexed definition of the cross-language relation $\propto$.

Theorem 8.1 (Fully Abstract Closure-conversion)
Let $\Delta; \Gamma \vdash e_1 : \tau \rightsquigarrow_{\mathrm{c}} \hat{e}_1$ *and* $\Delta; \Gamma \vdash e_2 : \tau \rightsquigarrow_{\mathrm{c}} \hat{e}_2$*. Then* $\Delta; \Gamma^{+} \vdash \hat{e}_1 \approx^{ctx} \hat{e}_2 : \tau^{+}$ *if and only if* $\Delta; \Gamma \vdash e_1 \approx^{ctx} e_2 : \tau$*.*

Proof: ($\Leftarrow$) Let $\hat{C}$ be an arbitrary, suitably typed target context. Consider $\hat{C}[\hat{e}_1]$ and $\hat{C}[\hat{e}_2]$: By Lemma 7.1 these are the same as $C'[e_1]$ and $C'[e_2]$ where $C' = \hat{C}[\mathcal{W}^{-}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{+}_{\llbracket\tau\rrbracket}([\cdot])]]$. Therefore, $e_1 \approx^{ctx} e_2$ implies $\hat{e}_1 \approx^{ctx} \hat{e}_2$.
($\Rightarrow$) Let C be an arbitrary, suitably typed source context. Consider $C[e_1]$ and $C[e_2]$: By Lemma 7.1 these are the same as $\hat{C}'[\hat{e}_1]$ and $\hat{C}'[\hat{e}_1]$ with $\hat{C}' = C[\mathcal{W}^{+}_{\llbracket\Gamma\rrbracket}[\mathcal{W}^{-}_{\llbracket\tau\rrbracket}([\cdot])]]$. Therefore, $\hat{e}_1 \approx^{ctx} \hat{e}_2$ implies $e_1 \approx^{ctx} e_2$. □

9. Discussion

Using a target language that is identical to the source language has potential disadvantages and advantages. The main disadvantage is the fact that by making the target language bigger than strictly necessary, the property to be proved becomes stronger because there are more contexts available that could be used to distinguish between translation terms, making the proof itself potentially harder. On the flip side, by making it possible to have terms of both the source and the target coexist as sub-terms of a larger expression, our wrapper technique becomes viable, offsetting the disadvantages of having to prove a stronger theorem.

Also, given a proof for the bigger target language, full abstraction for a smaller target is not immediate. However, preservation of equivalence *is* an immediate corollary. Let T_{small} be a sub-language of T_{large}. As explained in Section 2.2, if two T_{small}-terms are contextually equivalent in T_{large}, then they are also contextually equivalent in T_{small}, since T_{small} provides fewer contexts that could make distinguishing observations.

The other direction, reflection of equivalence, is not as immediate. However, it is also the easier direction, as it can be proved as a consequence of the *correctness* of the translation from source to T_{small}. Observe that for every source context C there exists a target (T_{small}) context C'—which can be obtained from C by an easy generalization of the translation mechanism to contexts—such that $C'[e']$ is the translation of $C[e]$ whenever e' is the translation of e. The rest now follows from correctness: if there exists a C that distinguishes between e_1 and e_2 at the source, then the corresponding C' will distinguish between e'_1 and e'_2 in T_{small}.

In the case where source and target are not identical, our general approach requires that we give a direct definition of the cross-language relation $\propto$. For the present case[7], this can be done in the same style in which we defined $\approx$, i.e., by defining a step-indexed binary relation. This involves defining *asymmetric* relations $\propto_{\leq}$ and $\propto_{\geq}$ whose intersection is then taken to be $\propto$. However, since left-hand and right-hand sides of $\propto$ belong to different languages (and/or have different types), $\propto_{\geq}$ cannot be obtained from $\propto_{\leq}$ by simply swapping the arguments. Each has to be defined separately.

We have sketched out a direct definition of $\propto$ in Figure 10 (only showing the most basic form of a function type with only a single value argument and no type parameters). Since $\propto$ is not symmetric, we define the two "halves" of the relation separately (cf., $\mathcal{T}^{\leq}_{\mathrm{C}}\,\llbracket\cdot\rrbracket$ and $\mathcal{T}^{\geq}_{\mathrm{C}}\,\llbracket\cdot\rrbracket$, respectively). Even if—as shown in the figure—we cut down on some of the notational duplication by abstracting over the "direction" in the definitions of $\mathcal{T}^{\leq}_{\mathrm{V}}\,\llbracket\cdot\rrbracket$ and $\mathcal{T}^{\geq}_{\mathrm{V}}\,\llbracket\cdot\rrbracket$, unifying them into the definition of $\mathcal{T}^{\xi}_{\mathrm{V}}\,\llbracket\cdot\rrbracket$, we are still faced with a formal framework several times larger than the one needed for the single-language plus wrappers scenario, because in addition to $\propto$ we also need to define $\approx_{\mathrm{S}}$ and $\approx_{\mathrm{T}}$ separately.

Approaching the full abstraction proof. After setting up the formalism, we are ready to state a number of lemmas that ultimately lead to a proof of equivalence-preserving and equivalence-reflecting translation.

First, there are two lemmas that connect $\propto$ to $\approx_{\mathrm{S}}$ and $\approx_{\mathrm{T}}$. The first of them states that $\propto$ respects equivalences both on the left and on the right:

Lemma 9.1
$\Delta; \Gamma \vdash e_1 \approx_{\mathrm{S}} e_2 : \tau \ \wedge\ \Delta; \Gamma \vdash e_1 \propto e' : \tau \ \Rightarrow\ \Delta; \Gamma \vdash e_2 \propto e' : \tau.$
$\Delta; \Gamma^{+} \vdash e'_1 \approx_{\mathrm{T}} e'_2 : \tau^{+} \ \wedge\ \Delta; \Gamma \vdash e \propto e'_1 : \tau \ \Rightarrow\ \Delta; \Gamma \vdash e \propto e'_2 : \tau.$

The other lemma expresses that being related to the same term via $\propto$ implies equivalence (both on the left and on the right):

Lemma 9.2
$\Delta; \Gamma \vdash e_1 \propto e' : \tau \ \wedge\ \Delta; \Gamma \vdash e_2 \propto e' : \tau \ \Rightarrow\ \Delta; \Gamma \vdash e_1 \approx_{\mathrm{S}} e_2 : \tau.$
$\Delta; \Gamma \vdash e \propto e'_1 : \tau \ \wedge\ \Delta; \Gamma \vdash e \propto e'_2 : \tau \ \Rightarrow\ \Delta; \Gamma^{+} \vdash e'_1 \approx_{\mathrm{T}} e'_2 : \tau^{+}.$

The last lemma justifies the construction of $\propto$ and is analogous to Lemma 7.1 by stating that it relates a source term to its translation:

Lemma 9.3 (Fundamental property of $\propto$)
If $\Delta; \Gamma \vdash e : \tau \rightsquigarrow \hat{e}$*, then* $\Delta; \Gamma \vdash e \propto \hat{e} : \tau$*.*

Obviously, the laborious part of the overall full abstraction proof is in the individual proofs for these five lemmas. It seems advisable to use the wrapper approach whenever possible, since it simplifies the formal setup and cuts down on the overall size of the proof. Nevertheless, if the wrapper approach cannot be made to work, the above proof outline might be a viable alternative, possibly using a different (but similar) setup for defining $\propto$ together with suitably modified Lemmas 9.1 and 9.3.

10. Conclusions and Related Work

Language-based security solutions that rely on programming language abstraction facilities have seen a considerable increase in popularity. Inherently, the approach relies on the assumption that

[7] i.e., ignoring for the moment that S and T are in fact the same language

all potential attacks are bound by the rules of the language in question. But programs are routinely translated (compiled) from one language to another. Thus, attacks can be launched at the level of a target language with different rules and weaker protection.

Nevertheless, for programmers it is much more convenient to reason about the behavior of their code in terms of the source language alone. This is justified if the translation process is *fully abstract*, meaning that no target context can make more observations than any source context. This is the problem of equivalence-preserving compilation.

In this paper, we have shown that typed closure conversion in the style of Morrisett et al. [21] is fully abstract. Recall from Section 2 that this also implies that the translation is correct (i.e., semantics-preserving). Proofs of correctness (but not of full abstraction) for functional closure conversion have been given by others (e.g., Minamide et al. [20]), although none of these results consider recursive types. Our proofs are based on operational techniques, in particular, a step-indexed logical relation. They do not involve game semantics or other denotational approaches.

Glew [12] showed a form of closure conversion for an object calculus and proved it fully abstract. But object closure conversion is simpler than functional closure conversion. Specifically, Glew notes that the latter can be encoded as the composition of (1) encoding functions as objects, (2) object closure conversion, and (3) an object encoding. Hence, for full abstraction of functional closure conversion, one would also need to prove encodings (1) and (3) fully abstract.

Riecke [27] investigates fully abstract translations between call-by-name, call-by-value, and lazy PCF. The proofs rely on fully abstract denotational models of the languages. Each of the languages includes the parallel conditional. This is needed to make the models fully abstract.

There has been a great deal of work on fully abstract denotational models of languages (e.g., [22, 19, 8, 13, 2]). Our emphasis is somewhat different in that we focus on type-directed and type-preserving compilation. Given a sufficiently "clever" type translation, the *types* of compiled terms can impose well-behavedness constraints on any target-level term that might interact with the result of the translation, thus ensuring that target contexts cannot violate source-level abstractions.

Our proof relies on the ability to "back-translate" target values into source values. As we have shown, in the setting where source language S and target language T are identical, this can be achieved by defining *wrappers*, i.e., terms *within the language* that implement the back-translation. In Section 2 and later in Section 9 we discussed the general outline of a different approach involving an explicitly defined cross-language relation $\propto$ between terms of S and T. However, we suspect that the details highly depend on the particular choice of S, T, and the translation $\leadsto$ between them. We are pessimistic about the existence of a general "framework" for proofs of full abstraction that can simply be "instantiated" to concrete S, T, and $\leadsto$.

In the future, we plan to prove a similar result for typed CPS conversion and later to extend our fully abstract compilation pipeline down to TAL. At the TAL level we will have to deal with state, pointers, and heaps. We hope that a TAL based on Hoare Type Theory [23] can provide the necessary parametricity properties in the presence of state that let us show such translations fully abstract.

References

[1] M. Abadi. Protection in programming-language translations. In *ICALP '98*, pages 868–883, London, UK, 1998.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. Technical Report TR-01-06, Harvard University, Mar. 2006. `ttic.uchicago.edu/~amal`.

[4] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP '06*, Mar. 2006.

[5] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. Technical Report TR-2008-07, Department of Computer Science, University of Chicago, July 2008.

[6] K. Arnold, J. Gosling, and D. Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley, 2005.

[7] M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Prog.*, 16(4-5):375–414, 2006.

[8] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *POPL '92*, pages 328–342, 1992.

[9] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, June 2005.

[10] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02*, pages 48–59. ACM Press, 2002.

[12] N. Glew. Object closure conversion. In *Higher-Order Operational Techniques in Semantics (HOOTS '99)*, Sept. 1999.

[13] A. Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *LICS '95*, 1995.

[14] A. Kennedy. Securing the .NET programming model. *Theor. Comput. Sci.*, 364(3):311–317, 2006.

[15] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06*, Jan. 2006.

[16] X. Leroy. Unboxed objects and polymorphic typing. In *POPL '92*, pages 177–188. ACM Press, Jan. 1992.

[17] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *J. Funct. Prog.*, 1(3):287–327, 1991.

[18] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL '07*, Jan. 2007.

[19] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL '88*, pages 191–203, 1988.

[20] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *POPL '96*, pages 271–283, Jan. 1996.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *POPL '98*, pages 85–97, Jan. 1998.

[22] K. Mulmuley. *Full abstraction and semantic equivalence*. MIT Press, 1987.

[23] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In *ESOP '07*, pages 189–204, 2007.

[24] A. M. Pitts. Existential types: Logical relations and operational equivalence. In *ICALP '98*, pages 309–326, 1998.

[25] G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI–RM–4, Univ. of Edinburgh, Oct. 1973.

[26] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[27] J. G. Riecke. Fully abstract translations between functional languages. In *POPL '91*, pages 245–254, 1991.

[28] Z. Shao. Flexible representation analysis. In *ICFP '97*, pages 85–98. ACM Press, 1997.

[29] W. W. Tait. Intensional interpretations of functionals of finite type I. *J. of Symbolic Logic*, 32(2):198–212, June 1967.