

Equation-preserving multilanguage systems

Jacob Matthews
University of Chicago,
jacobm@cs.uchicago.edu

Abstract

Kennedy has recently argued that the designers of C^\sharp should try to informally verify that their compilation from C^\sharp source to Microsoft Common Language Runtime (CLR) bytecode is “fully abstract” in the sense that any two code fragments that C^\sharp programs cannot distinguish must compile into CLR code fragments that CLR programs also cannot distinguish. In this paper we treat that property as a criterion for what constitutes a well-designed interoperability system and refine it using the techniques of Matthews-Findler-style multilanguage systems. We begin by giving our own variation on Kennedy’s criterion that does not directly involve a notion of compilation. Then we argue that our criterion is useful and workable by using it to guide our development of a multilanguage system that combines call-by-name and call-by-value lambda calculi. We begin with a straightforward system that we show does not meet our criterion. Then we refine that system’s interoperability rules and show that the result does meet our criterion. We conclude by sketching how our criterion would apply to multilanguage systems whose constituent languages had more advanced features such as state, exceptions, and concurrent threads.

1 What makes a foreign interface design good?

Kennedy [1], following Abadi [2], argues that we need to think about interoperability as a full abstraction problem: Given source language S and target language T (with contextual equivalence relations \cong_s and \cong_t) and translation function $\mathcal{T} : S \rightarrow T$ that compiles the source into the target, he makes the case that we should try to establish that for any source-language terms e_1 and e_2 ,

$$\begin{array}{ccc} e_1 & \cong_s & e_2 \\ & \Downarrow & \\ \mathcal{T}(e_1) & \cong_t & \mathcal{T}(e_2) \end{array}$$

His reasoning is that programmers — or at least, expert programmers — rely on their intuitive understanding of a programming language’s contextual equivalence relation in building secure abstractions, and if his property held it would “ensure that C^\sharp programmers [could] reason about the security of their code by *thinking in C^\sharp* ” (emphasis in original). Conversely, places where it fails represent potential security risks since a language’s defense mechanisms might be overridden.

This property is important to more than just security. It applies equally well to compiler optimization, refactoring, and any other domain in which we might want to take advantage of program equivalence in a language that has any kind of foreign interface. For this reason, we view Kennedy’s property as getting at a ‘gold standard’ criterion that separates good interoperability facilities from bad: a good foreign interface allows programmers to forget about it, while a bad one does not. To take an obvious example, Haskell’s foreign function interface is ‘good’ under this criterion because it uses the IO monad to sequester impure foreign function calls; if it did not, then such calls would disturb its equivalence relation and break programmers’ ability to forget about them.

Kennedy’s specific formulation in terms of fully abstract translations, though, is unwieldy for this purpose. First, it necessarily involves a compiler \mathcal{T} ; in our setting we may not even have a compiler to reason about, and even if we do we are more interested in the semantics of the languages involved than how they are implemented. Second, purely psychologically, it suggests that in order to prove your language implementation secure you have to prove a universal property about all assembly-language programs that might get linked in with your compiled code; in many interoperability contexts, though, both interacting languages have richer equational properties that we may be able to exploit.

In this paper we suggest a variation that builds on our previous with multilanguage systems [3]. Instead of using source and target languages S and T and the translation function \mathcal{T} , we imagine two peer languages L and K and multilanguage system LK (with contextual equivalence relation \cong_{l+k}) which includes both L and K connected by boundaries. In this setting, the gold standard is that equivalence in source-language contexts implies equivalence in arbitrary contexts:

$$\begin{array}{ccc} e_1 & \cong_l & e_2 \\ & \Downarrow & \\ e_1 & \cong_{l+k} & e_2 \end{array}$$

This formulation is actually stronger than Kennedy’s in the sense that we could pick C^\sharp for L and the .NET intermediate language for K , and then if our formulation holds then Kennedy’s also holds for any semantics-preserving translation function \mathcal{T} . Furthermore it has the nice property that rather than focusing our attention on the translation, it focuses our attention on the semantics of a computation flowing across boundaries, and in particular it calls to attention that appropriate boundaries may be able to defend against malicious foreign contexts.

In the rest of this paper, we argue our formulation’s utility by pursuing an example: forming a multilanguage embedding that combines a simply-typed call-by-name language with a simply-typed call-by-value language. We begin with a straightforward ‘natural’ embedding of the two languages in the style of an embedding from prior work [3, section 3]. We show by the method of logical relations that we can compile the multilanguage system into a single language using Plotkin’s call-by-name and call-by-value CPS transformations [4]; then we show that, nonetheless, there are pairs of terms that are contextually equivalent when considering only call-by-name contexts that can be distinguished by contexts that contain call-by-value portions. Considering one example leads us to a slight refinement of our semantic model and a somewhat

| | |
|---|---|
| $ \begin{aligned} \mathbf{e} &= x \mid (\mathbf{e} \mathbf{e}) \mid \lambda x. \mathbf{e} \mid \bar{n} \mid \uparrow \\ \mathbf{v} &= \lambda x. \mathbf{e} \mid \mathbf{Nat} \\ \mathbf{E} &= []_N \mid (\mathbf{E} \mathbf{e}) \end{aligned} $ $ \frac{(\mathbf{x} : \tau) \in \Gamma \quad \Gamma \vdash_n \mathbf{e}_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_n \mathbf{e}_2 : \tau_1}{\Gamma \vdash_n \mathbf{x} : \tau \quad \Gamma \vdash_n (\mathbf{e}_1 \mathbf{e}_2) : \tau_2} \quad \frac{\Gamma, (\mathbf{x} : \tau_1) \vdash_n \mathbf{e} : \tau_2}{\Gamma \vdash_n \lambda \mathbf{x}. \mathbf{e} : \tau_1 \rightarrow \tau_2} \quad \frac{}{\Gamma \vdash_n \bar{n} : \mathbb{N}} \quad \frac{}{\Gamma \vdash_n \uparrow : \tau} $ $ \begin{aligned} \mathcal{E}[(\lambda x. \mathbf{e}_1) \mathbf{e}_2]_N &\mapsto \mathcal{E}[\mathbf{e}_1\{x := \mathbf{e}_2\}] \\ \mathcal{E}[\uparrow]_N &\mapsto \uparrow \end{aligned} $ | $ \begin{aligned} \mathbf{e} &= x \mid (\mathbf{e} \mathbf{e}) \mid \lambda x. \mathbf{e} \mid \mathbf{Nat} \mid \uparrow \\ \mathbf{v} &= \lambda x. \mathbf{e} \mid \mathbf{Nat} \\ \mathbf{E} &= []_V \mid (\mathbf{E} \mathbf{e}) \mid (\mathbf{v} \mathbf{E}) \end{aligned} $ $ \frac{(\mathbf{x} : \tau) \in \Gamma \quad \Gamma \vdash_v \mathbf{e}_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_v \mathbf{e}_2 : \tau_1}{\Gamma \vdash_v \mathbf{x} : \tau \quad \Gamma \vdash_v (\mathbf{e}_1 \mathbf{e}_2) : \tau_2} \quad \frac{\Gamma, (\mathbf{x} : \tau_1) \vdash_v \mathbf{e} : \tau_2}{\Gamma \vdash_v \lambda \mathbf{x}. \mathbf{e} : \tau_1 \rightarrow \tau_2} \quad \frac{}{\Gamma \vdash_v \bar{n} : \mathbb{N}} \quad \frac{}{\Gamma \vdash_v \uparrow : \tau} $ $ \begin{aligned} \mathcal{E}[(\lambda x. \mathbf{e}) \mathbf{v}]_V &\mapsto \mathcal{E}[\mathbf{e}\{x := \mathbf{v}\}] \\ \mathcal{E}[\uparrow]_V &\mapsto \uparrow \end{aligned} $ |
| $\mathcal{E} = \mathbf{E}$ | |

Figure 1: Call-by-name (left) and call-by-value (right) languages

larger change to our compilation strategy, after which we prove that the refined version does satisfy our criterion. We conclude by sketching several other examples of multi-language systems involving more sophisticated language features whose designs could be informed by our criterion.

2 A first system

In this section we present a first attempt at a multilanguage system combining call-by-name and call-by-value. We show that it satisfies a number of nice properties but that it does not satisfy the gold standard.

2.1 Syntax and reduction rules

Figure 1 defines grammars and reduction rules for our call-by-name and call-by-value languages (hereafter Λ_n and Λ_v), with Λ_n on the left in bold red font and Λ_v on the right in sans-serif blue font. Each system is standard except that we give each language a single effect represented as the term \uparrow which inhabits all types and immediately terminates the program when evaluated (it is intended to suggest a diverging computation). We add \uparrow only so that we can write programs whose outcomes depend on evaluation order, which would be impossible otherwise since all programs would terminate and evaluation order does not change the final answer of a program that terminates. The languages are both simply-typed with the base type \mathbb{N} (for “natural number”) and arrow type $\tau_1 \rightarrow \tau_2$. We use $\Gamma \vdash_n \mathbf{e} : \tau$ for Λ_n ’s typing judgment and $\Gamma \vdash_v \mathbf{e} : \tau$ for Λ_v ’s typing

judgment¹. We assume that we can syntactically differentiate variables introduced by call-by-name terms and variables introduced by call-by-value terms.

As a first pass at combining the two languages, we extend the language grammars to include boundaries by adapting the natural embedding of prior work [3, section 3] as closely as possible. We extend nonterminals \mathbf{e} and \mathbf{e} such that $\mathbf{e} = \dots \mid (\mathbf{NV}^\tau \mathbf{e})$ and $\mathbf{e} = \dots \mid (\mathbf{VN}^\tau \mathbf{e})$ (here we use the \dots notation to indicate that we are not removing the productions listed in figure 1 for these nonterminals, only adding new ones). Similarly we extend evaluation contexts \mathbf{E} and \mathbf{E} such that $\mathbf{E} = \dots \mid (\mathbf{NV}^\tau \mathbf{E})$ and $\mathbf{E} = \dots \mid (\mathbf{VN}^\tau \mathbf{E})$. Third we add typing rules. Since both sides have the same type system, the typing rules at boundaries are straightforward:

$$\frac{\Gamma \vdash_v \mathbf{e} : \tau}{\Gamma \vdash_n (\mathbf{NV}^\tau \mathbf{e}) : \tau} \quad \frac{\Gamma \vdash_n \mathbf{e} : \tau}{\Gamma \vdash_v (\mathbf{VN}^\tau \mathbf{e}) : \tau}$$

The last step of defining the embedding is to add new reduction rules for the new syntactic forms:

$$\begin{array}{ll} \mathcal{E}[\mathbf{NV}^{\mathbb{N}} n]_N & \mapsto \mathcal{E}[n] & (\text{NVNum}) \\ \mathcal{E}[\mathbf{NV}^{\tau_1 \rightarrow \tau_2} \mathbf{v}]_N & \mapsto \mathcal{E}[\lambda x. (\mathbf{NV}^{\tau_2} (\mathbf{v} (\mathbf{VN}^{\tau_1} x)))] & (\text{NVFun}) \\ \\ \mathcal{E}[\mathbf{VN}^{\mathbb{N}} n]_V & \mapsto \mathcal{E}[n] & (\text{VNNum}) \\ \mathcal{E}[\mathbf{VN}^{\tau_1 \rightarrow \tau_2} \mathbf{v}]_V & \mapsto \mathcal{E}[\lambda x. (\mathbf{VN}^{\tau_2} (\mathbf{v} (\mathbf{NV}^{\tau_1} x)))] & (\text{VNFun}) \end{array}$$

Here we have chosen straightforward reduction rules for boundaries that mirror the behavior of boundaries we have seen so far. We call the language formed this way $\Lambda_{(n+v)_1}$.

Theorem 2.1 (type soundness). $\Lambda_{(n+v)_1}$ is type-sound.

Before we proceed, we should also take a moment to define the notion of contextual equivalence that we use throughout the paper. For a given language Λ_x , we will define contexts to be terms in language x “that have a hole in them”, *i.e.* a single subtree removed. If a context C would have type τ under environment Γ if its hole were replaced by a term of language Λ_y that had type τ' under environment Γ' , we write $C : (\Gamma' \vdash_y \tau') \rightsquigarrow (\Gamma \vdash_x \tau)$. We define whole-program contexts to be those contexts with type $C : (\Gamma \vdash_y \tau) \rightsquigarrow (\vdash_x \mathbb{N})$ for any Γ, y , and τ , and we define a contextual equivalence relation as follows:

Definition 2.2 (Kleene equivalence). We say that $a =_{\text{kleene}} b$ iff $a \mapsto_x^* \uparrow \Leftrightarrow b \mapsto_x^* \uparrow$ and $a \mapsto_x^* \bar{n} \Leftrightarrow b \mapsto_x^* \bar{n}$.

Definition 2.3 (contextual equivalence). We say $\Gamma \vdash a \cong_x b : \tau$ iff for all contexts C of Λ_x such that $C : (\Gamma \vdash_x \tau) \rightsquigarrow (\vdash_x \mathbb{N})$, we have that $C[a] =_{\text{kleene}} C[b]$.

¹To be very precise we could define isomorphic but separate alphabets, say τ and σ , for the types given to Λ_n and Λ_v languages respectively. This would add considerably to the weight of our notation, though, without providing any additional insight, so we choose to use the same types for both languages.

$$\begin{array}{lcl}
\llbracket \tau \rrbracket_N & = & ([\tau]_N \rightarrow \sigma) \rightarrow \sigma \\
\llbracket \mathbb{N} \rrbracket_N & = & \mathbb{N} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_N & = & \llbracket \tau_1 \rrbracket_N \rightarrow \llbracket \tau_2 \rrbracket_N \\
\llbracket x \rrbracket_N & = & x \\
\llbracket n \rrbracket_N & = & \lambda k. k n \\
\llbracket \uparrow \rrbracket_N & = & \lambda k. \uparrow \\
\llbracket \lambda x. e \rrbracket_N & = & \lambda k. k (\lambda x. \llbracket e \rrbracket_N) \\
\llbracket (e_1 e_2) \rrbracket_N & = & \lambda k. (\llbracket e_1 \rrbracket_N (\lambda f. (f \llbracket e_2 \rrbracket_N k))) \\
\llbracket \mathbf{VN}^\tau e \rrbracket_N & = & (\mathcal{W}_{NV}^\tau \llbracket e \rrbracket_V) \\
\llbracket \tau \rrbracket_V & = & ([\tau]_V \rightarrow \sigma) \rightarrow \sigma \\
\llbracket \mathbb{N} \rrbracket_V & = & \mathbb{N} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_V & = & \llbracket \tau_1 \rrbracket_V \rightarrow \llbracket \tau_2 \rrbracket_V \\
\llbracket x \rrbracket_V & = & \lambda k. k x \\
\llbracket n \rrbracket_V & = & \lambda k. k n \\
\llbracket \uparrow \rrbracket_V & = & \lambda k. \uparrow \\
\llbracket \lambda x. e \rrbracket_V & = & \lambda k. k (\lambda x. \llbracket e \rrbracket_V) \\
\llbracket (e_1 e_2) \rrbracket_V & = & \lambda k. \llbracket e_1 \rrbracket_V (\lambda f. \llbracket e_2 \rrbracket_V (\lambda x. f x k)) \\
\llbracket \mathbf{VN}^\tau e \rrbracket_V & = & (\mathcal{W}_{VN}^\tau \llbracket e \rrbracket_N)
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{NV}^\tau : \llbracket \tau \rrbracket_N \rightarrow \llbracket \tau \rrbracket_V \\
\mathcal{W}_{NV}^\mathbb{N} = \lambda c. c \\
\mathcal{W}_{NV}^{\tau_1 \rightarrow \tau_2} = \\
\lambda c. \lambda k. (c (\lambda f. (k (\lambda x. (\mathcal{W}_{NV}^{\tau_2} (f (\mathcal{W}_{NV}^{\tau_1} (\lambda k'. k' x)))))))) \\
\mathcal{W}_{VN}^\tau : \llbracket \tau \rrbracket_V \rightarrow \llbracket \tau \rrbracket_N \\
\mathcal{W}_{VN}^\mathbb{N} = \lambda c. c \\
\mathcal{W}_{VN}^{\tau_1 \rightarrow \tau_2} = \\
\lambda c. \lambda k. (c (\lambda f. (k (\lambda c'. \lambda k'. (\mathcal{W}_{NV}^{\tau_1} c') (\lambda x. (\mathcal{W}_{VN}^{\tau_2} (f x))) k'))))
\end{array}$$

Figure 2: Boundary-elimination CPS transformation 1

2.2 Eliminating boundaries with CPS

$\Lambda_{(n+v)_1}$ is an intuitively simple way to describe an interaction between Λ_n and Λ_v that corresponds to an interpreter. We can also derive a semantically-equivalent compiler from it. To do so we combine both of Plotkin’s original continuation-passing-style transformations [4] in a single system; as one might hope, Plotkin’s rules map over directly and to make the system work all we need to do is define translations for boundaries.

Figure 2 defines three related pieces that together form the CPS translation $\llbracket \cdot \rrbracket_N$ (for Λ_n source terms) and $\llbracket \cdot \rrbracket_V$ (for Λ_v source terms). Those pieces are the CPS type translation, the CPS term translation, and definitions for a type-indexed family of “wrapper” terms \mathcal{W}_{NV}^τ and \mathcal{W}_{VN}^τ used by the term translation for boundary cases.

The type translations $\llbracket \tau \rrbracket_N$ and $\llbracket \tau \rrbracket_V$ give the types for Λ_n and Λ_v computations of type τ , while $[\tau]_N$ and $[\tau]_V$ give the types for CPS-converted Λ_n and Λ_v values of type τ . Notice that while the term translations are functions, the type relations are not: any concrete type can be chosen for the abstract “answer” type σ . We choose to encode answer types this way so that we can avoid including polymorphic types in the

languages themselves; in effect we are keeping the universal quantifier in the meta-level rather than encoding it at the term level. As a matter of notation, when we write types involving $\llbracket \tau \rrbracket_N$ or $\llbracket \tau \rrbracket_V$ (for instance, $\llbracket \tau \rrbracket_N \rightarrow \llbracket \tau \rrbracket_V$) we mean to indicate the *set* of types that can be formed by choosing a concrete answer type for σ . When we ascribe a term such a set of types (for instance, $\mathcal{W}_{NV}^\tau : \llbracket \tau \rrbracket_N \rightarrow \llbracket \tau \rrbracket_V$) we mean that the given term can be typed regardless of which concrete type σ represents.

The term translations are Plotkin's call-by-name and call-by-value CPS transformations, each extended with a translation for boundaries. These cases have the same simple pattern: they take a foreign computation and translate it using the appropriate wrapper function to a native computation. For the base case, note that $\llbracket \mathbb{N} \rrbracket_N$ and $\llbracket \mathbb{N} \rrbracket_V$ represent the same concrete type (*i.e.*, $(\mathbb{N} \rightarrow \sigma) \rightarrow \sigma$) and CPS terms of that type have the same concrete behavior (to yield an integer to the continuation) the wrappers at type \mathbb{N} can be the identity. The wrappers for arrow types both essentially return a new computation that forces the input computation to a value and then returns a function value that performs the appropriate wrappings on its inputs and outputs.

The three portions of figure 2 are connected to each other by the following theorem, which says that for any term e of type τ , $\llbracket e \rrbracket_N$ has type $\llbracket \tau \rrbracket_N$ and similarly for Λ_V terms:

Theorem 2.4 (CPS type-correctness). *Taking $\llbracket \Gamma \rrbracket$ to mean $\{(x : \llbracket \tau \rrbracket_N) \mid (x : \tau) \in \Gamma\} \cup \{(x : \llbracket \tau \rrbracket_V) \mid (x : \tau) \in \Gamma\}$, both of the following hold:*

1. *If $\Gamma \vdash_n e : \tau$ then $\llbracket \Gamma \rrbracket \vdash_n \llbracket e \rrbracket_N : \llbracket \tau \rrbracket_N$*
2. *If $\Gamma \vdash_v e : \tau$ then $\llbracket \Gamma \rrbracket \vdash_v \llbracket e \rrbracket_V : \llbracket \tau \rrbracket_V$*

The CPS translation is also connected to the operational semantics we gave previously by the fact that it is semantics-preserving: a program that terminates with a particular integer value under the original semantics compiles into a program that terminates with the same integer value, and a program that reduces to \uparrow in the original semantics compiles into another program that reduces to \uparrow .

To show this we use the method of logical relations. First we define a logical relation $e_{\text{src}} \sim_c e_{\text{cps}} : \tau$ between terms. The left side of the relation is intended for source-language terms and comprises both Λ_n and Λ_V terms. To remind the reader of this we annotate variables representing these terms with subscript *src* (for instance, e_{src}). The right side is intended for Λ_n terms in continuation-passing style. To suggest this we annotate variables representing these terms with subscript *cps* (for instance, e_{cps}). The logical relation is in figure 3.

As is often done with logical relations, we extend the relation to term substitutions with the relation $\gamma_{\text{src}} \sim_c \gamma_{\text{cps}} : \Gamma$:

Definition 2.5 (related term substitutions). *We say that $\gamma_{\text{src}} \sim_c \gamma_{\text{cps}} : \Gamma$ iff:*

1. *For each $(x : \tau) \in \Gamma$, there exist terms e_{src} and e_{cps} such that $\gamma_{\text{src}}(x) = e_{\text{src}}$, $\gamma_{\text{cps}}(x) = e_{\text{cps}}$, and $e_{\text{src}} \sim_c e_{\text{cps}} : \tau$.*
2. *For each $(x : \tau) \in \Gamma$, there exist terms v_{src} and e_{cps} such that $\gamma_{\text{src}}(x) = v_{\text{src}}$, $\gamma_{\text{cps}}(x) = e_{\text{cps}}$, and $v_{\text{src}} \sim_c e_{\text{cps}} : \tau$.*

$$\begin{aligned}
\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau &\stackrel{\text{def}}{=} \vdash_n \mathbf{e}_{\text{src}} : \tau, \vdash_n \mathbf{e}_{\text{cps}} : \llbracket \tau \rrbracket_N \\
&\text{and } \mathbf{e}_{\text{src}} \hookrightarrow^* \uparrow \text{ and } \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* \uparrow \\
&\text{or } \mathbf{e}_{\text{src}} \hookrightarrow^* \mathbf{v}_{\text{src}}, \\
&\quad \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* (k \mathbf{v}_{\text{cps}}), \\
&\text{and } \mathbf{v}_{\text{src}} \sim_{\text{c}, \nu} \mathbf{v}_{\text{cps}} : \tau \\
\\
n_{\text{src}} \sim_{\text{c}, \nu} n_{\text{cps}} : \mathbb{N} &\stackrel{\text{def}}{=} n_{\text{src}} = n_{\text{cps}} \\
\mathbf{v}_{\text{src}} \sim_{\text{c}, \nu} \mathbf{v}_{\text{cps}} : \tau_1 \rightarrow \tau_2 &\stackrel{\text{def}}{=} \forall \mathbf{e}'_{\text{src}}, \mathbf{e}'_{\text{cps}} \\
&\text{where } \mathbf{e}'_{\text{src}} \sim_{\text{c}} \mathbf{e}'_{\text{cps}} : \tau_1 \\
&\text{we have } (\mathbf{v}_{\text{src}} \mathbf{e}'_{\text{src}}) \sim_{\text{c}} \lambda k. (\mathbf{v}_{\text{cps}} \mathbf{e}'_{\text{cps}} k) : \tau_2 \\
\\
\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau &\stackrel{\text{def}}{=} \vdash_{\nu} \mathbf{e}_{\text{src}} : \tau, \vdash_n \mathbf{e}_{\text{cps}} : \llbracket \tau \rrbracket_V, \\
&\text{and } \mathbf{e}_{\text{src}} \hookrightarrow^* \uparrow \text{ and } \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* \uparrow \\
&\text{or } \mathbf{e}_{\text{src}} \hookrightarrow^* \mathbf{v}_{\text{src}}, \\
&\quad \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* (k \mathbf{v}_{\text{cps}}), \\
&\text{and } \mathbf{v}_{\text{src}} \sim_{\text{c}, \nu} \mathbf{v}_{\text{cps}} : \tau \\
\\
n_{\text{src}} \sim_{\text{c}, \nu} n_{\text{cps}} : \mathbb{N} &\stackrel{\text{def}}{=} n_{\text{src}} = n_{\text{cps}} \\
\mathbf{v}_{\text{src}} \sim_{\text{c}, \nu} \mathbf{v}_{\text{cps}} : \tau_1 \rightarrow \tau_2 &\stackrel{\text{def}}{=} \forall \mathbf{v}'_{\text{src}}, \mathbf{v}'_{\text{cps}} \\
&\text{where } \mathbf{v}'_{\text{src}} \sim_{\text{c}, \nu} \mathbf{v}'_{\text{cps}} : \tau_1 \\
&\text{we have } (\mathbf{v}_{\text{src}} \mathbf{v}'_{\text{src}}) \sim_{\text{c}} \lambda k. (\mathbf{v}_{\text{cps}} \mathbf{v}'_{\text{cps}} k) : \tau_2
\end{aligned}$$

Figure 3: CPS logical relation 1

With that, we are ready to prove that any well-typed term is related to its own CPS conversion:

Lemma 2.6 (CPS-correctness lemma 1). *Both of the following are true:*

1. *If $\Gamma \vdash_n \mathbf{e} : \tau$ and $\gamma_{\text{src}} \sim_{\text{c}} \gamma_{\text{cps}} : \Gamma$, then $\gamma_{\text{src}}(\mathbf{e}) \sim_{\text{c}} \gamma_{\text{cps}}(\llbracket \mathbf{e} \rrbracket_N) : \tau$*
2. *If $\Gamma \vdash_{\nu} \mathbf{e} : \tau$ and $\gamma_{\text{src}} \sim_{\text{c}} \gamma_{\text{cps}} : \Gamma$, then $\gamma_{\text{src}}(\mathbf{e}) \sim_{\text{c}} \gamma_{\text{cps}}(\llbracket \mathbf{e} \rrbracket_V) : \tau$*

Lemma 2.7 (CPS-correctness bridge lemma 1). *Both of the following hold:*

1. *If $\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau$ then $(\mathcal{V}N^{\tau} \mathbf{e}_{\text{src}}) \sim_{\text{c}} (\mathcal{W}_{\mathcal{V}N}^{\tau} \mathbf{e}_{\text{cps}}) : \tau$.*
2. *If $\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau$ then $(\mathcal{N}V^{\tau} \mathbf{e}_{\text{src}}) \sim_{\text{c}} (\mathcal{W}_{\mathcal{N}V}^{\tau} \mathbf{e}_{\text{cps}}) : \tau$.*

Theorem 2.8 (compiler correctness 1). *If $\vdash_n \mathbf{e} : \mathbb{N}$ then $\mathbf{e} =_{\text{kleene}} (\llbracket \mathbf{e} \rrbracket_N (\lambda I. I))$.*

Proof. Special case of lemma 2.6. □

2.3 Equivalence preservation

So far we have shown $\Lambda_{(n+v)_1}$ to be type-sound and we have given a compiler that reduces it to pure- Λ_n terms using the CPS transformation. Unfortunately, despite these facts, it does not satisfy our gold standard. For a counterexample, consider the two terms \uparrow and $(\lambda x. \uparrow)$. It is known that $\vdash \uparrow \cong_n (\lambda x. (\uparrow x)) : \mathbb{N} \rightarrow \mathbb{N}$ (though this depends on the fact that our notion of contextual equivalence considers only program contexts whose outer type is \mathbb{N}). Now consider the context $\mathbf{C}_1 = (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\mathbf{VN}^{\mathbb{N} \rightarrow \mathbb{N}} [\]_N)))$. It is easy to check that $\mathbf{C}_1 : (\vdash_n \mathbb{N} \rightarrow \mathbb{N}) \rightsquigarrow (\vdash_n \mathbb{N})$ and thus \mathbf{C}_1 is a program context. Plugging the two terms in, we have

$$\begin{aligned} \mathbf{C}_1[\uparrow] &= (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\mathbf{VN}^{\mathbb{N} \rightarrow \mathbb{N}} \uparrow))) \\ &\mapsto \uparrow \\ \mathbf{C}_1[\lambda x. \uparrow] &= (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\mathbf{VN}^{\mathbb{N} \rightarrow \mathbb{N}} \lambda x. \uparrow))) \\ &\mapsto (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\lambda y. \mathbf{NV}^{\mathbb{N}} ((\lambda x. \uparrow) (\mathbf{VN}^{\mathbb{N}} y)))) \mapsto 12 \end{aligned}$$

Hence $\Gamma \vdash \mathbf{e}_1 \cong_n \mathbf{e}_2 : \tau$ does not imply that $\Gamma \vdash \mathbf{e}_1 \cong_{(n+v)_1} \mathbf{e}_2 : \tau$.

3 Another choice for boundaries

As we have shown, the boundaries we defined in section 2 are not equation-preserving and thus $\Lambda_{(n+v)_1}$ does not satisfy the gold standard. Our counterexample suggests where the embedding's extra observational power comes from: since boundaries force their contents on evaluation, they can be used to directly observe termination at arrow types, an observation that is otherwise impossible. Using this intuition, we can build an alternate set of boundaries that are equation-preserving.

3.1 Syntax and reduction rules

Instead of the extensions to the core languages made in section 2, we instead make the extensions of figure 4 to form $\Lambda_{(n+v)_2}$. Evaluation contexts (listed on the third and fourth lines of that figure) are now sensitive to the type at a boundary: they still directly force boundaries that contain call-by-name terms of type \mathbb{N} , but do not evaluate inside boundaries that contain call-by-name terms with arrow types. Instead, the last listed reduction rule directly reduces a \mathbf{VN} boundary with an arrow type to a function value without forcing the term it contains.

3.2 Eliminating boundaries with CPS

As before, we can eliminate boundaries by performing a CPS transformation of the program. The transformation is defined as before, but with new wrapping procedures as in figure 5. Also as before we can establish that this translation is semantics-preserving through a logical relation between source terms and their translations. The relation is nearly the same as before, except that we modify the definition of the relation at call-by-name function types to relate source and CPS terms if all possible applications

$$\begin{aligned}
\mathbf{e} &= \dots | (\mathbf{NV}^\tau \mathbf{e}) \\
\mathbf{e} &= \dots | (\mathbf{VN}^\tau \mathbf{e}) \\
\mathbf{E} &= (\mathbf{NV}^\tau \mathbf{E}) \\
\mathbf{E} &= (\mathbf{VN}^\mathbb{N} \mathbf{E})
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\mathbf{NV}^\mathbb{N} n]_N &\mapsto \mathcal{E}[n] \\
\mathcal{E}[\mathbf{NV}^{\tau_1 \rightarrow \tau_2} \lambda x. \mathbf{e}]_N &\mapsto \mathcal{E}[\lambda x. (\mathbf{NV}^{\tau_2} ((\lambda x. \mathbf{e}) (\mathbf{VN}^{\tau_1} x)))] \\
\mathcal{E}[\mathbf{NV}^{\tau_1 \rightarrow \tau_2} (\mathbf{VN}^{\tau_1 \rightarrow \tau_2} \mathbf{e})]_N &\mapsto \mathcal{E}[\mathbf{e}] \\
\mathcal{E}[\mathbf{VN}^\mathbb{N} n]_V &\mapsto \mathcal{E}[n] \\
\mathcal{E}[(\mathbf{VN}^{\tau_1 \rightarrow \tau_2} \mathbf{e})]_V &\mapsto \mathcal{E}[\lambda x. (\mathbf{VN}^{\tau_2} (\mathbf{e} (\mathbf{NV}^{\tau_1} x)))]
\end{aligned}$$

Figure 4: Extensions to form equation-preserving system

to related terms are related, regardless of whether the source term's function position coterminates with the CPS term's function computation. The relation is defined in figure 6.

Lifting that relation to apply to substitution using the same procedure as in definition 2.5, we prove correctness using the same method as before:

Lemma 3.1 (CPS-correctness lemma 2). *Both of the following hold:*

1. *If $\Gamma \vdash_n \mathbf{e} : \tau$ and $\gamma_{\text{src}} \sim_{c_2} \gamma_{\text{cps}} : \Gamma$, then $\gamma_{\text{src}}(\mathbf{e}) \sim_{c_2} \gamma_{\text{cps}}(\llbracket \mathbf{e} \rrbracket_N) : \tau$*
2. *If $\Gamma \vdash_v \mathbf{e} : \tau$ and $\gamma_{\text{src}} \sim_{c_2} \gamma_{\text{cps}} : \Gamma$, then $\gamma_{\text{src}}(\mathbf{e}) \sim_{c_2} \gamma_{\text{cps}}(\llbracket \mathbf{e} \rrbracket_V) : \tau$*

Lemma 3.2 (CPS-correctness bridge lemma 2). *Both of the following hold:*

1. *If $\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau$, then $(\mathbf{VN}^\tau \mathbf{e}_{\text{src}}) \sim_{c_2} (\mathcal{W}_{\mathbf{NV}}^\tau \mathbf{e}_{\text{cps}}) : \tau$.*
2. *If $\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau$, then $(\mathbf{NV}^\tau \mathbf{e}_{\text{src}}) \sim_{c_2} (\mathcal{W}_{\mathbf{VN}}^\tau \mathbf{e}_{\text{cps}}) : \tau$.*

Theorem 3.3 (compiler correctness 2). *If $\vdash_n \mathbf{e} : \mathbb{N}$ then $\mathbf{e} =_{\text{kleene}} (\llbracket \mathbf{e} \rrbracket_N (\lambda I. I))$.*

3.3 Equivalence preservation

We can develop a notion of equivalence in $\Lambda_{(n+v)_2}$ again by means of logical relations, this time using the relation defined in figure 7. We establish that this relation is sound with respect to contextual equivalence in the combined language by establishing that it is a consistent, congruent, reflexive relation. Consistency is immediate:

Lemma 3.4 (consistency). *If $\mathbf{e}_1 \sim_n \mathbf{e}_2 : \mathbb{N}$ then $\mathbf{e}_1 =_{\text{kleene}} \mathbf{e}_2$.*

Proof. Immediate by the definition of logical equivalence. □

Next we prove that reflexivity holds:

$$\begin{array}{l|l}
\llbracket x \rrbracket_N & = x \\
\llbracket n \rrbracket_N & = \lambda k. k n \\
\llbracket \uparrow \rrbracket_N & = \lambda k. \uparrow \\
\llbracket \lambda x. e \rrbracket_N & = \lambda k. k (\lambda x. \llbracket e \rrbracket_N) \\
\llbracket (e_1 e_2) \rrbracket_N & = \\
& \lambda k. (\llbracket e_1 \rrbracket_N (\lambda f. (f \llbracket e_2 \rrbracket_N k))) \\
\llbracket \mathbf{NV}^\tau e \rrbracket_N & = (\mathcal{W}_{NV}^\tau \llbracket e \rrbracket_V) \\
\hline
\llbracket x \rrbracket_V & = \lambda k. k x \\
\llbracket n \rrbracket_V & = \lambda k. k n \\
\llbracket \uparrow \rrbracket_V & = \lambda k. \uparrow \\
\llbracket \lambda x. e \rrbracket_V & = \lambda k. k (\lambda x. \llbracket e \rrbracket_V) \\
\llbracket (e_1 e_2) \rrbracket_V & = \\
& \lambda k. \llbracket e_1 \rrbracket_V (\lambda f. \llbracket e_2 \rrbracket_V (\lambda x. f x k)) \\
\llbracket \mathbf{VN}^\tau e \rrbracket_V & = (\mathcal{W}_{VN}^\tau \llbracket e \rrbracket_N)
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{NV}^N & = \lambda c. c \\
\mathcal{W}_{NV}^{\tau_1 \rightarrow \tau_2} & = \\
\lambda z. \lambda k_1. (k_1 (\lambda x_1. (\mathcal{W}_{NV}^{\tau_2} (\lambda k_2. (z (\lambda v_1. (v_1 (\mathcal{W}_{VN}^{\tau_1} (\lambda k_3. (k_3 x_1))) k_2))))))) &
\end{array}$$

$$\begin{array}{l}
\mathcal{W}_{VN}^N & = \lambda c. c \\
\mathcal{W}_{VN}^{\tau_1 \rightarrow \tau_2} & = \\
\lambda z. \lambda k_1. z (\lambda f. (k_1 (\lambda x_1. \lambda k_2. ((\mathcal{W}_{NV}^{\tau_1} x_1) (\lambda v_1. (\mathcal{W}_{VN}^{\tau_2} (f v_1) k_2)))))) &
\end{array}$$

Figure 5: Boundary-elimination CPS transformation 2

Lemma 3.5 (reflexivity / fundamental theorem of the logical relation). *Both of the following are true:*

1. *If $\Gamma \vdash_n e : \tau$, then $\Gamma \vdash e \sim_n e : \tau$*
2. *If $\Gamma \vdash_v e : \tau$, then $\Gamma \vdash e \sim_v e : \tau$*

The proof of this lemma is standard except that it requires a bridge lemma that establishes that relation in one language implies relation in the other:

Lemma 3.6 (reflexivity bridge lemma). *Both of the following are true:*

1. *If $\Gamma \vdash e_1 \sim_n e_2 : \tau$, then $\Gamma \vdash \mathbf{VN}^\tau e_1 \sim_v \mathbf{VN}^\tau e_2 : \tau$*
2. *If $\Gamma \vdash e_1 \sim_v e_2 : \tau$, then $\Gamma \vdash \mathbf{NV}^\tau e_1 \sim_n \mathbf{NV}^\tau e_2 : \tau$*

We use this lemma to prove congruence, which is otherwise standard, though note that we must unfortunately check all combinations of call-by-name or call-by-value contexts with call-by-name or call-by-value holes. (This may seem strange at first, but notice that $(\mathbf{NV}^\tau []_V)$ and $(\mathbf{VN}^\tau []_N)$ are contexts whose top-level language and hole language differ.)

Lemma 3.7 (congruence). *All of the following are true:*

1. *If $\Gamma \vdash e_1 \sim_n e_2 : \tau$ and $C : (\Gamma \vdash_n \tau) \rightsquigarrow (\Gamma' \vdash_n \tau')$ then $\Gamma' \vdash C[e_1] \sim_n C[e_2] : \tau'$*
2. *If $\Gamma \vdash e_1 \sim_v e_2 : \tau$ and $C : (\Gamma \vdash_v \tau) \rightsquigarrow (\Gamma' \vdash_v \tau')$ then $\Gamma' \vdash C[e_1] \sim_v C[e_2] : \tau'$*

$$\begin{aligned}
\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \mathbb{N} &\stackrel{\text{def}}{=} \text{either } \mathbf{e}_{\text{src}} \hookrightarrow^* \uparrow \text{ and } \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* \uparrow \\
&\text{or } \mathbf{e}_{\text{src}} \hookrightarrow^* \mathbf{v}_{\text{src}}, \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* (k \mathbf{v}_{\text{cps}}) \\
&\text{and } \exists n \in \mathbb{N}. \mathbf{v}_{\text{src}} = \mathbf{v}_{\text{cps}} = \bar{n} \\
\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau_1 \rightarrow \tau_2 &\stackrel{\text{def}}{=} \forall \mathbf{e}'_{\text{src}}, \mathbf{e}'_{\text{cps}} \text{ such that } \mathbf{e}'_{\text{src}} \sim_{c_2} \mathbf{e}'_{\text{cps}} : \tau_1. \\
&(\mathbf{e}_{\text{src}} \mathbf{e}'_{\text{src}}) \sim_{c_2} (\lambda k. (\mathbf{e}_{\text{cps}} (\lambda f. (f \mathbf{e}'_{\text{cps}} k)))) : \tau_2 \\
\\
\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau &\stackrel{\text{def}}{=} \text{either } \mathbf{e}_{\text{src}} \hookrightarrow^* \uparrow \text{ and } \forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* \uparrow \\
&\text{or } \mathbf{e}_{\text{src}} \hookrightarrow^* \mathbf{v}_{\text{src}}, \\
&\forall k. (\mathbf{e}_{\text{cps}} k) \hookrightarrow^* (k \mathbf{v}_{\text{cps}}), \\
&\text{and } \mathbf{v}_{\text{src}} \sim_{c_2, \nu} \mathbf{v}_{\text{cps}} : \tau \\
\\
n_{\text{src}} \sim_{c_2, \nu} n_{\text{cps}} : \mathbb{N} &\stackrel{\text{def}}{=} n_{\text{src}} = n_{\text{cps}} \\
\mathbf{v}_{\text{src}} \sim_{c_2, \nu} \mathbf{v}_{\text{cps}} : \tau_1 \rightarrow \tau_2 &\stackrel{\text{def}}{=} \forall \mathbf{v}'_{\text{src}}, \mathbf{v}'_{\text{cps}} \\
&\text{where } \mathbf{v}'_{\text{src}} \sim_{c_2, \nu} \mathbf{v}'_{\text{cps}} : \tau_1 \\
&\text{we have } (\mathbf{v}_{\text{src}} \mathbf{v}'_{\text{src}}) \sim_{c_2} \lambda k. \mathbf{v}_{\text{cps}} \mathbf{v}'_{\text{cps}} k : \tau_2
\end{aligned}$$

Figure 6: CPS logical relation 2

3. If $\Gamma \vdash \mathbf{e}_1 \sim_{\nu} \mathbf{e}_2 : \tau$ and $\mathbf{C} : (\Gamma \vdash_{\nu} \tau) \rightsquigarrow (\Gamma' \vdash_{\nu} \tau')$ then $\Gamma' \vdash \mathbf{C}[\mathbf{e}_1] \sim_{\nu} \mathbf{C}[\mathbf{e}_2] : \tau'$
4. If $\Gamma \vdash \mathbf{e}_1 \sim_{\nu} \mathbf{e}_2 : \tau$ and $\mathbf{C} : (\Gamma \vdash_{\nu} \tau) \rightsquigarrow (\Gamma' \vdash_{\nu} \tau')$ then $\Gamma' \vdash \mathbf{C}[\mathbf{e}_1] \sim_{\nu} \mathbf{C}[\mathbf{e}_2] : \tau'$

With these lemmas all established, we can prove the theorem of interest.

Theorem 3.8. *Both of the following are true:*

1. If $\mathbf{e}_1 \sim_n \mathbf{e}_2 : \tau$ then $\mathbf{e}_1 \cong_{n+\nu} \mathbf{e}_2 : \tau$
2. If $\mathbf{e}_1 \sim_{\nu} \mathbf{e}_2 : \tau$ then $\mathbf{e}_1 \cong_{n+\nu} \mathbf{e}_2 : \tau$

Proof. Special case of lemma 3.7. □

In section 2.3 we showed that \uparrow and $(\lambda x. \uparrow)$, which are equivalent in Λ_n , could be distinguished using the $\Lambda_{(n+\nu)_1}$ context $\mathbf{C}_1 = (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\mathbf{VN}^{\tau_1 \rightarrow \tau_2} []_N)))$. Under $\Lambda_{(n+\nu)_2}$, however, \mathbf{C}_1 no longer distinguishes the terms:

$$\begin{aligned}
\mathbf{C}_1[\uparrow] &= (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\mathbf{VN}^{\tau_1 \rightarrow \tau_2} \uparrow))) \\
&\mapsto (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\lambda x. (\mathbf{VN}^{\tau_2} (\uparrow (\mathbf{NV}^{\tau_1} x)))))) \mapsto (\mathbf{NV}^{\mathbb{N}} 12) \mapsto 12 \\
\\
\mathbf{C}_1[(\lambda x. \uparrow)] &= (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\mathbf{VN}^{\tau_1 \rightarrow \tau_2} (\lambda x. \uparrow)))) \\
&\mapsto (\mathbf{NV}^{\mathbb{N}} ((\lambda f. 12) (\lambda x. (\mathbf{VN}^{\tau_2} ((\lambda x. \uparrow) (\mathbf{NV}^{\tau_1} x)))))) \mapsto (\mathbf{NV}^{\mathbb{N}} 12) \mapsto 12
\end{aligned}$$

In fact, using theorem 3.8 we can see that those two terms are not distinguishable by any $\Lambda_{(n+\nu)_2}$ contexts at all:

$$\begin{array}{lcl}
\mathbf{e}_1 \sim_n \mathbf{e}_2 : \mathbb{N} & \stackrel{\text{def}}{=} & \exists \mathbf{v}_1, \mathbf{v}_2 \in \Lambda_{(n+v)_1}. \\
& & \forall \mathcal{E}. \mathcal{E}[\mathbf{e}_1] \mapsto^* \mathcal{E}[\mathbf{v}_1] \Leftrightarrow \mathcal{E}[\mathbf{e}_2] \mapsto^* \mathcal{E}[\mathbf{v}_2] \text{ and } \mathbf{v}_1 = \mathbf{v}_2 \\
\mathbf{e}_1 \sim_n \mathbf{e}_2 : \tau_1 \rightarrow \tau_2 & \stackrel{\text{def}}{=} & \forall \mathbf{e}'_1, \mathbf{e}'_2 \in \Lambda_{(n+v)_1}. \\
& & \mathbf{e}'_1 \sim_n \mathbf{e}'_2 : \tau_1 \Rightarrow (\mathbf{e}_1 \mathbf{e}'_1) \sim_n (\mathbf{e}_2 \mathbf{e}'_2) : \tau_2 \\
\mathbf{e}_1 \sim_v \mathbf{e}_2 : \tau & \stackrel{\text{def}}{=} & \exists \mathbf{v}_1, \mathbf{v}_2 \in \Lambda_{(n+v)_1}. \\
& & \forall \mathcal{E}. \mathcal{E}[\mathbf{e}_1] \mapsto^* \mathbf{v}_1 \Leftrightarrow \mathcal{E}[\mathbf{e}_2] \mapsto^* \mathbf{v}_2 \text{ and } \mathbf{v}_1 \sim_v \mathbf{v}_2 : \tau \\
\mathbf{v}_1 \sim_v \mathbf{v}_2 : \mathbb{N} & \stackrel{\text{def}}{=} & \mathbf{v}_1 = \mathbf{v}_2 \\
\mathbf{v}_1 \sim_v \mathbf{v}_2 : \tau_1 \rightarrow \tau_2 & \stackrel{\text{def}}{=} & \forall \mathbf{v}'_1, \mathbf{v}'_2 \in \Lambda_{(n+v)_1}. \\
& & \mathbf{v}'_1 \sim_v \mathbf{v}'_2 : \tau_1 \Rightarrow (\mathbf{v}_1 \mathbf{v}'_1) \sim_v (\mathbf{v}_2 \mathbf{v}'_2) : \tau_2
\end{array}$$

Figure 7: Logical equivalence relation for terms in the second embedding

Proposal 3.9. $\uparrow \cong_{n+v} \lambda x. \uparrow : \mathbf{Nat} \rightarrow \mathbf{Nat}$.

Proof. Fix $\mathbf{e}_1, \mathbf{e}_2$ such that $\mathbf{e}_1 \simeq_n \mathbf{e}_2 : \mathbf{Nat}$. We have that $(\uparrow \mathbf{e}_1) \mapsto \uparrow$ and $((\lambda x. \uparrow) \mathbf{e}_2) \mapsto \uparrow$. Hence by definition of the logical relation, $\uparrow \sim_n \lambda x. \uparrow : \mathbf{Nat} \rightarrow \mathbf{Nat}$. Thus by theorem 3.8, $\uparrow \cong_{n+v} \lambda x. \uparrow : \mathbf{Nat} \rightarrow \mathbf{Nat}$ as required. \square

4 The gold standard

The fact that proposal 3.9 holds is a good sign, but to establish the gold standard we need to show that *any* two terms that are equal in Λ_n alone are also equal in $\Lambda_{(n+v)_2}$.

Our strategy is to prove the gold standard's contrapositive, *i.e.* that if there exists a $\Lambda_{(n+v)_2}$ program context that distinguishes two purely- Λ_n terms, then there is also a Λ_n context that distinguishes them². We can almost show this by construction: suppose $\mathbf{C} []_N$ is a $\Lambda_{(n+v)_2}$ program context. It is equivalent by a single beta-reduction to the context $((\lambda t. \mathbf{C}[t] []_N)$ (where t does not appear in \mathbf{C}). We would like to now use the CPS translation turn this into a pure-call-by-name term something like $((\lambda t. \llbracket \mathbf{C}[t] \rrbracket_N []_N (\lambda I. I)))$ — to adapt it to open terms we could simply perform the equivalent of lambda-lifting and generalize the context to

$$((\lambda t. \llbracket \mathbf{C}[(t x_1 \dots)] \rrbracket_N) (\lambda x_1 \dots. []_N) (\lambda I. I)))$$

²At this point a reader may wonder why this result is not a near-immediate implication of the fact that the relation given in figure 7 corresponds to contextual equivalence given that its two halves look very similar to equivalence relations for separate call-by-name and call-by-value languages. The answer is that the similarity is only superficial. Consider the two cases for arrow types: in a standard logical equivalence relation, two call-by-name (or call-by-value) terms are related at arrow types if their applications to related *purely call-by-name* (or purely call-by-value) arguments are related. In our relation, though, two call-by-name (or call-by-value) terms are related at arrow types if their applications to related call-by-name terms that *potentially contain call-by-value boundaries* (or call-by-value terms that potentially contain call-by-name boundaries) are related. We can think of this section as an attempt to show that the latter implies the former.

where $x_1 \dots$ is a list of all variables bound by the context at the point where the hole occurs.

This strategy does not quite work, though, because it would mean we were substituting arbitrary terms into contexts that were expecting CPS computations. Fortunately, though, we can dynamically convert from a plain call-by-name term into an equivalent CPS computation (and vice versa, as turns out to be necessary) using the following type-indexed functions that are defined directly as call-by-name terms:

$$\begin{aligned}
exp^\tau & : \tau \rightarrow \llbracket \tau \rrbracket_N \\
exp^{\mathbb{N}} & \stackrel{\text{def}}{=} \lambda x. \lambda k. k(x + 0) \\
exp^{\tau_1 \rightarrow \tau_2} & \stackrel{\text{def}}{=} \lambda f. \lambda k. k(\lambda x. (exp^{\tau_2} (f (imp^{\tau_1} x)))) \\
\\
imp^\tau & : \llbracket \tau \rrbracket_N \rightarrow \tau \\
imp^{\mathbb{N}} & \stackrel{\text{def}}{=} \lambda x. (x (\lambda I. I)) \\
imp^{\tau_1 \rightarrow \tau_2} & \stackrel{\text{def}}{=} \lambda c. c(\lambda f. \lambda x. (imp^{\tau_2} (f (exp^{\tau_1} x))))
\end{aligned}$$

We establish that these functions are correct by proving that if two terms e_1 and e_2 are logically equivalent (under the combined call-by-name and call-by-value equivalence relation) at type τ , then e_1 is CPS-related to $(exp^\tau e_2)$ at type τ and simultaneously that if e_1 is CPS-related to e_2 at type τ then e_1 is logically equivalent to $(imp^\tau e_2)$ at type τ .

Lemma 4.1 (exp^τ and imp^τ correctness). *Both of the following hold:*

1. *If $e_1 \sim_n e_2 : \tau$, then $e_1 \sim_{c_2} (exp^\tau e_2) : \tau$.*
2. *If $e_{\text{src}} \sim_{c_2} e_{\text{cps}} : \tau$, then $e_{\text{src}} \sim_n (imp^\tau e_{\text{cps}}) : \tau$.*

Given exp^τ , we use the context $((\lambda x. (\llbracket \mathbf{C}[x] \rrbracket_N (\lambda I. I))) (exp^\tau []_N))$ (henceforth \mathbf{C}') to complete the proof outline above. By construction, $\mathbf{C}'[e]$ terminates with a number \bar{n} if and only if $\mathbf{C}[e]$ terminates. Furthermore, it is a purely call-by-name context. Since we have placed no restrictions on \mathbf{C} , we can build this device for an arbitrary context. The following theorem codifies that intuitive argument.

Lemma 4.2 (The gold standard contrapositive). *If $\Gamma \vdash e_1 \not\sim_{n+v} e_2 : \tau$ where e_1 and e_2 are entirely call-by-name terms, then $\Gamma \vdash e_1 \not\sim_n e_2 : \tau$.*

Proof. By definition, $\exists \mathbf{C} : (\Gamma \vdash \tau) \rightsquigarrow (\vdash \mathbf{N})$ such that $\mathbf{C}[e_1]$ disagrees with $\mathbf{C}[e_2]$. Furthermore, by beta-equivalence this context is equivalent to $((\lambda x. \mathbf{C}[(x x_1 \dots)]) (\lambda x_1 \dots. []_N))$ where $x_1 \dots$ indicates a sequence consisting of the variables in Γ in some canonical order (we will use $\tau_1 \dots$ as the sequence consisting of those variables' types). By lemma 3.1, we have that $(\lambda x. \mathbf{C}[(x x_1 \dots)]) \sim_{c_2} \llbracket (\lambda x. \mathbf{C}[(x x_1 \dots)]) \rrbracket_N : (\tau_1 \dots \rightarrow \tau) \rightarrow \mathbf{N}$. Applying lemma 3.5 gives us that $(\lambda x_1 \dots. e_1) \sim_n (\lambda x_1 \dots. e_1) : \tau_1 \dots \rightarrow \tau$ and hence by lemma 4.1 we have that $(\lambda x_1 \dots. e_1) \sim_{c_2} (exp^{\tau_1 \dots \rightarrow \tau} (\lambda x_1 \dots. e_1)) : \tau_1 \dots \rightarrow \tau$. Applying the definition of the CPS logical relation, we have that

$$e'_n \sim_{c_2} e'_{\text{cps}} : \mathbf{N}$$

where

$$\begin{aligned} \mathbf{e}'_n &= ((\lambda t. \mathbf{C}[(t x_1 \dots)]) (\lambda x_1 \dots. \mathbf{e}_1)) \\ \mathbf{e}'_{cps} &= \lambda k. (\llbracket (\lambda x. \mathbf{C}[(x x_1 \dots)]) \rrbracket_N (\lambda f. f (exp^{\tau_1 \dots \tau_n \rightarrow \tau} (\lambda x_1 \dots. \mathbf{e}_1))) k) \end{aligned}$$

Thus by definition if $\mathbf{C}[\mathbf{e}_1] \mapsto^* \bar{n}$ then the CPSed version is a computation that reduces to the same number when applied to $\lambda I. I$, and furthermore if $\mathbf{C}[\mathbf{e}_1] \mapsto^* \uparrow$ then the CPSed version does as well when applied to $\lambda I. I$. Hence we have that

$$\mathbf{C}[\mathbf{e}_1] =_{\text{kleene}} \lambda k. (\llbracket (\lambda t. \mathbf{C}[(t x_1 \dots)]) \rrbracket_N (\lambda f. f (exp^{\tau_1 \dots \tau_n \rightarrow \tau} (\lambda x_1 \dots. \mathbf{e}_1))) k) (\lambda I. I)$$

The same line of reasoning applies to $\mathbf{C}[\mathbf{e}_2]$; thus

$$\mathbf{C}[\mathbf{e}_2] =_{\text{kleene}} \lambda k. (\llbracket (\lambda t. \mathbf{C}[(t x_1 \dots)]) \rrbracket_N (\lambda f. f (exp^{\tau_1 \dots \tau} (\lambda x_1 \dots. \mathbf{e}_2))) k) (\lambda I. I)$$

and hence

$$\begin{aligned} &\lambda k. (\llbracket (\lambda t. \mathbf{C}[(t x_1 \dots)]) \rrbracket_N (\lambda f. f (exp^{\tau_1 \dots \tau} (\lambda x_1 \dots. \mathbf{e}_1))) k) (\lambda I. I) \\ &\quad \neq_{\text{kleene}} \lambda k. (\llbracket (\lambda t. \mathbf{C}[(t x_1 \dots)]) \rrbracket_N (\lambda f. f (exp^{\tau_1 \dots \tau} (\lambda x_1 \dots. \mathbf{e}_2))) k) (\lambda I. I) \end{aligned}$$

Therefore the context

$$\lambda k. (\llbracket (\lambda t. \mathbf{C}[(t x_1 \dots)]) \rrbracket_N (\lambda f. f (exp^{\tau_1 \dots \tau} (\lambda x_1 \dots. []_N))) k) (\lambda I. I)$$

distinguishes \mathbf{e}_1 and \mathbf{e}_2 . This is an entirely call-by-name context; thus $\Gamma \vdash \mathbf{e}_1 \not\cong_n \mathbf{e}_2 : \tau$. \square

Theorem 4.3 (The gold standard). *For entirely call-by-name terms \mathbf{e}_1 and \mathbf{e}_2 , $\Gamma \vdash \mathbf{e}_1 \cong_n \mathbf{e}_2 : \tau$ if and only if $\Gamma \vdash \mathbf{e}_1 \cong_{n+v} \mathbf{e}_2 : \tau$.*

Proof. The “if” direction is a corollary of lemma 4.2; the “only if” direction holds trivially because entirely call-by-name contexts are a subset of call-by-name plus call-by-value contexts. \square

It is important to point out that though we have used CPS and thus a “compilation-based” argument in support of the gold standard claim, the result is a property that is completely independent of any compiler.

5 Related work

At the beginning of this article we discussed our relation to Kennedy. Felleisen [6, theorems 3.9 and 3.15] establishes that Λ_n cannot macro-express call-by-value lambda abstractions. This does not contradict our results in section 3: in Felleisen’s system, call-by-value lambda abstractions always force their arguments, exactly the problem that caused $\Lambda_{(n+v)_1}$ to fail. He also notes that while a programming language extension that can tell apart two otherwise indistinguishable terms necessarily adds to the language’s expressive power, it is possible for extensions that add expressive power to preserve the base language’s equations [6, theorem 3.14ii]. Our proposed gold standard,

then, suggests that well-designed foreign interfaces should either be macro-expressible in the host language or fall into this second case.

[7] gives another result (theorem 15) that at first glance appears to contradict the development we have presented here. Again, there is no contradiction, but this time for a more subtle reason that points at the additional insight we gain from the Matthews-Findler multilanguage system framework. The translations Riecke considers must be performed in a vacuum: the translation of any term must behave in an observably identical way to the original term and be insertable into an arbitrary foreign context. Our translations do not behave this way; $\llbracket \tau \rrbracket_N$ and $\llbracket \tau \rrbracket_V$ produce computations that are only suitable for composing with other computations of the same kind. Call-by-name computations, but not call-by-value computations, can be made implicit with imp^τ in a meaning-preserving way, and computations can be converted from one language to another using \mathcal{W}_{NV}^τ and \mathcal{W}_{VN}^τ but these conversions behave as though they had boundaries around them. (The technical version of this point is that it does not appear that within our system there exist $proj^\tau$ and inj^τ functions that satisfy the constraints of Riecke’s definition 13 — certainly neither exp^τ and imp^τ nor $\mathcal{W}_{NV}^\tau \circ exp^\tau$ and $imp^\tau \circ \mathcal{W}_{VN}^\tau$ fit the bill — thus it appears that our system is not a “functional translation” as per his definition, and thus theorem 15 does not apply.)

6 Conclusions and further applications

We believe that the process we have been through in this paper — choose two languages, connect them with boundaries, and then vary those boundaries’ semantics until you find semantics that preserve the gold standard — is an effective way to design foreign interfaces between general-purpose languages, even if you do not establish the gold standard formally. To illustrate this, we conclude by sketching designs for multilanguage systems involving languages with different features. In each example, we describe a naive embedding that we argue does not preserve the gold standard, then describe another embedding we argue does preserve it.

6.1 State

Suppose that language L is a strict language with no imperative state features. In L , we have that $(\lambda fg. g (f 1) (f 1)) \cong_l \lambda fg. \mathbf{let } y = f 1 \mathbf{ in } g y y \mathbf{ end}$: even in the presence of nontermination or error signaling, in this language running $(f 1)$ and using its result twice is the same as running $(f 1)$ twice. Now suppose that K is another language that has the features of L plus mutable state (in this example, using ML-style ref cells).

Under a naive embedding, the single language equation does not hold in multilanguage contexts, *i.e.* $(\lambda fg. g (f 1) (f 1)) \not\cong_{l+k} \lambda fg. \mathbf{let } y = f 1 \mathbf{ in } g y y \mathbf{ end}$. To see why, consider the context

$$(\mathbf{let } x = \mathbf{ref } 0 \mathbf{ in } ((\mathbf{KL } []) (\lambda z. (x := !x + 1; !x)) (\lambda ab. a == b)) \mathbf{end})$$

This context provides a function that returns a different value each time it is called, and uses it to observe the difference between the two terms.

This can be fixed in at least two ways, both of which involve restricting the types of boundaries. The first way is to restrict boundary types monadically *a la* Haskell, so that any function that enters L from K consumes and produces an additional “world state” argument. Under this scheme enemy context we are thinking of contains a boundary whose type is illegal and is not a counterexample. The second way is to add linear types to L [5] and add a global value and make the type of every boundary linear; this would prevent any boundary from being used more than once which would again make our proposed counterexample ill-typed.

6.2 Exceptions

Suppose that language L now has mutable state. In this language it seems clear that $(\lambda f. \mathbf{let } x = \mathbf{ref } 0 \mathbf{ in } (f(\lambda g. (x := 1; g()); x := 0)); !x) \mathbf{end}$ is contextually equivalent to $(\lambda f. \mathbf{let } x = \mathbf{ref } 0 \mathbf{ in } (f(\lambda g. (x := 1; g()); x := 0)); 0) \mathbf{end}$. In other words, suppose we have a ref cell whose initial value is 0 and function that takes a thunk g and does nothing other set x to 1, call g for effect, and then set x back to 0. If we provide that function to some arbitrary context-chosen function f (that has no access to x), then no matter what f does the value of x after it returns (if it returns at all) will be 0.

But if K contains exceptions, then in a naive embedding this equation does not hold, *i.e.* $\lambda f. \mathbf{let } x = \mathbf{ref } 0 \mathbf{ in } (f(\lambda g. (x := 1; g()); x := 0)); !x) \mathbf{end}$ is not contextually equal to $\lambda f. \mathbf{let } x = \mathbf{ref } 0 \mathbf{ in } (f(\lambda g. (x := 1; g()); x := 0)); 0) \mathbf{end}$. This is because a context that can throw and catch exceptions can arrange to skip parts of the stack, for instance with the context

$$((\mathbf{KL } []) (\lambda h. \mathbf{try } h(\lambda _. \mathbf{throw } ()) \mathbf{catch } e \Rightarrow ()))$$

By having g throw an exception and then catching it, f causes x to be incremented but not decremented.

This can be fixed by making boundaries catch exceptions and then either aborting the program or converting them into values. This strategy appears to allow the attacker context without allowing it to distinguish the two terms.

6.3 Thread systems

Finally, imagine that L now has mutable state and exceptions. We have that

$$\lambda x. (x := 1; !x) \cong_l \lambda x. (x := 1; 1)$$

but if it is embedded naively into a language K with threads, then it is possible to distinguish $\lambda x. (x := 1; !x)$ and $\lambda x. (x := 1; 1)$. This is because it introduces a race condition, as could be exposed by the context

$$\mathbf{let } x = (\mathbf{KL } \mathbf{ref } 0) \mathbf{ in } (\mathbf{spawn}(\mathbf{KL } \lambda _. x := 2); (\mathbf{KL } [] (\mathbf{LK } x))) \mathbf{end}$$

which might set x to 2 after the two terms set x to 1 but before they return either $!x$ or 1.

We can fix this by demanding that whenever L runs it does so in a single-threaded mode (which can be encoded formally by restricting evaluation contexts on boundaries) or less conservatively by only allowing a single thread at a time to evaluate L code.

References

- [1] Kennedy, A.: Securing the .NET programming model. *Theoretical Computer Science* **364**(3) (2006) 311–317
- [2] Abadi, M.: Protection in programming-language translations. In: *International Conference on Automata, Languages and Programming*. (1998)
- [3] Matthews, J., Fidler, R.B.: Operational semantics for multi-language programs. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. (2007) Extended version appears as University of Chicago Technical Report TR-2007-8, under review.
- [4] Plotkin, G.D.: Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science* **1** (1975) 125–159
- [5] Wadler, P.: Linear types can change the world! In Broy, M., Jones, C., eds.: *IFIP TC 2 Working Conference on Programming Concepts and Methods*. (1990)
- [6] Felleisen, M.: On the expressive power of programming languages. *Science of Computer Programming* **17** (1991) 35–75
- [7] Riecke, J.G.: Fully abstract translations between functional languages. In: *POPL: The 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. (1991)