1 Introduction

Gradually typed languages

2 Source Language Syntax

We take as our source language GTFL, a gradually-typed functional language with *evidence*. Such a language is used as the result of elaboration in the framework of Abstracting Gradual Typing (AGT) [Garcia et al., 2016], and allows the meaning of gradual programs to be determined in terms of evidence.

2.1 Terms

The syntax for terms for GTFL is given in Figure 1. The language is essentially a simply typed lambda calculus with integers, booleans, and base types, except that a term e may be ascribed with *evidence* epsilon. This evidence contains typing information that evolves throughout the run of a program. We explain evidence in detail in subsection 2.3. Because gradual typing may result in dynamic type errors, we have a special failure term **error**.

Values are defined in the usual way, except that we do not allow a value to be ascribed multiple pieces of evidence. To enforce this syntactically, we separate *raw values* (using the metavariable r), which do not contain evidence at the top-level, from general values (metavariable v), which ascribe a raw value with zero or one pieces of evidence.

While the original presentation of AGT treated terms as intrinsically typed values, we adopt the simpler approach used by Toro et al. [2019], where evidence ascription is included in the syntax for terms.

$n \in \mathbb{Z}, b \in \mathbb{B}$

$$
\begin{array}{llll}
\mathsf{e} & ::= & & \\
& \mid & x & \text{Variables} \\
& \mid & b & \text{Booleans} \\
& \mid & n & \text{Natural Numbers} \\
& \mid & \lambda x.\, \mathsf{e} & \text{Functions} \\
& \mid & \mathsf{e}_1\ \mathsf{e}_2 & \text{Function Application} \\
& \mid & \mathsf{e}_1 + \mathsf{e}_2 & \text{Addition} \\
& \mid & \mathsf{e}_1 \stackrel{?}{=} \mathsf{e}_2 & \text{Number Equality Test} \\
& \mid & \mathbf{if}\ \mathsf{e}_1\ \mathbf{then}\ \mathsf{e}_2\ \mathbf{else}\ \mathsf{e}_3 & \text{Conditionals} \\
& \mid & \langle \mathsf{e}_1, \mathsf{e}_2 \rangle & \text{Tuples} \\
& \mid & \pi_1 \mathsf{e} & \text{Tuple First Projection} \\
& \mid & \pi_2 \mathsf{e} & \text{Tuple Second Projection} \\
& \mid & \varepsilon\ \mathsf{e} & \text{Evidence Ascription} \\
& \mid & \mathbf{error} & \text{Runtime Type Error}
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{v} & ::= & & \text{Irreducable (closed) terms} \\
& \mid & \varepsilon\ \mathsf{r} & \\
& \mid & b & \\
& \mid & n & \\
& \mid & \lambda x.\, \mathsf{e} & \text{Functions} \\
& \mid & \langle \mathsf{v}_1, \mathsf{v}_2 \rangle &
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{r} & ::= & & \text{Raw Irreducable (closed) terms} \\
& \mid & b & \\
& \mid & n & \\
& \mid & \lambda x.\, \mathsf{e} & \text{Functions} \\
& \mid & \langle \mathsf{v}_1, \mathsf{v}_2 \rangle &
\end{array}
$$

Figure 1: Source Language Syntax: Terms

2.2 Types

As a gradually-typed language, the interesting features of GTFL are in its type system. The syntax for types, given in Figure 2, matches what one expects in a simply-typed calculus, except that we have also introduced the *unknown* or *dynamic* type $?$. Any term can have be assigned type $?$, and a term of type $?$ can be used in any context without being rejected as ill-typed.

To define our typing rules in subsection 2.3, we need *contexts*, which assign types to free program variables. Having types also allows us to precisely define what evidence is: each piece of evidence is simply a type. For the term $\varepsilon\ \mathsf{e}$, ε represents the most precise type knowledge we currently have about e, though as we will see below, ε may not exactly match the type we treat e as having.

$$
\begin{array}{llll}
\tau & ::= & & \text{Types} \\
 & | & \mathsf{Nat} & \\
 & | & \mathsf{Bool} & \\
 & | & \tau_1 {\rightarrow} \tau_2 & \\
 & | & \tau_1 {\times} \tau_2 & \\
 & | & ? &
\end{array}
$$

$$
\begin{array}{llll}
\Gamma & ::= & & \text{Environments} \\
 & | & \cdot & \\
 & | & \Gamma, (x : \tau) &
\end{array}
$$

$$
\begin{array}{lll}
\varepsilon & ::= & \\
 & | & \{\tau\}
\end{array}
$$

Figure 2: Source Language Syntax: Types

2.3 Static Semantics

$\boxed{\Gamma \vdash \mathsf{e} : \tau}$ *(Typability relation)*

$$\textsc{HasTypeVar}\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{HasTypeBool}\ \frac{}{\Gamma \vdash b : \mathsf{Bool}} \qquad \textsc{HasTypeNat}\ \frac{}{\Gamma \vdash n : \mathsf{Nat}}$$

$$\textsc{HasTypePlus}\ \frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{Nat}}{\Gamma \vdash \mathsf{e}_1 {+} \mathsf{e}_2 : \mathsf{Nat}} \qquad \textsc{HasTypeEq}\ \frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{Nat}}{\Gamma \vdash \mathsf{e}_1 \stackrel{?}{=} \mathsf{e}_2 : \mathsf{Nat}}$$

$$\textsc{HasTypeLam}\ \frac{\Gamma, (x : \tau_1) \vdash \mathsf{e} : \tau_2}{\Gamma \vdash \lambda x.\, \mathsf{e} : \tau_1 {\rightarrow} \tau_2} \qquad \textsc{HasTypeApp}\ \frac{\Gamma \vdash \mathsf{e}_1 : \tau_1 {\rightarrow} \tau_2 \qquad \Gamma \vdash \mathsf{e}_2 : \tau_1}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \tau_2}$$

$$\textsc{HasTypeIf}\ \frac{\Gamma \vdash \mathsf{e} : \mathsf{Bool} \qquad \Gamma \vdash \mathsf{e}_1 : \tau \qquad \Gamma \vdash \mathsf{e}_2 : \tau}{\Gamma \vdash \textbf{if}\ \mathsf{e}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2 : \tau}$$

$$\textsc{HasTypePair}\ \frac{\Gamma \vdash \mathsf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathsf{e}_2 : \tau_2}{\Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \tau_1 {\times} \tau_2} \qquad \textsc{HasTypeProj}\ \frac{\Gamma \vdash \mathsf{e} : \tau_1 {\times} \tau_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{p}_i\, \mathsf{e} : \tau}$$

$$\textsc{HasTypeAscr}\ \frac{\Gamma \vdash \mathsf{e} : \tau_2 \qquad \varepsilon \vdash \tau_1 \cong \tau_2}{\Gamma \vdash \varepsilon\, \mathsf{e} : \tau_1}$$

Figure 3: Source Language: Type Rules

The typing rules for our language are given in Figure 3. We assume that terms in this language are the result of a combined type-checking and elaboration pass. Because of this, the typing rules are not syntax directed, but mainly establish the safety of the language [Garcia et al., 2016]. Once again, the typing rules are entirely standard, except for HastypeAscr. This rule says that if e has type τ_2, we give $\varepsilon\ \mathsf{e}$ type τ_1 provided that ε is evidence that τ_1 and τ_2 are *consistent*.

We define consistency in terms of the precision meet operator: two types are consistent provided that some third type is more precise than both of them (Figure 4). We write $\varepsilon \vdash \tau_1 \cong \tau_2$ to mean that ε is evidence of the consistency of τ_1 and τ_2. Such a relationship holds whenever ε is at least as precise as both τ_1 and τ_2.

The meet operator itself is defined in Figure 4. We wish $?$ to be consistent with any type, so $?$ acts as an identity for the meet operator. The meet of τ with itself is τ, and the meet of function or arrow types is computed using the meet of the component types. Note that this is not subtyping: there is no contravariance in the rule for arrow types.

Including a notion of consistency in our type system allows us to type terms that would be ill-typed in a fully-static language. For example, $1 + \{\mathsf{Nat}\}\ (\{\mathsf{Bool}\}\ \mathbf{true})$ is well-typed in our language: $\cdot \vdash \mathbf{true} : \mathsf{Bool}$, and $\{\mathsf{Bool}\} \vdash \mathsf{Bool} \cong ?$, so $\cdot \vdash \{\mathsf{Bool}\}\ \mathbf{true} : ?$. Similarly, $\{\mathsf{Nat}\} \vdash ? \cong \mathsf{Nat}$, so $\cdot \vdash \{\mathsf{Nat}\}\ (\{\mathsf{Bool}\}\ \mathbf{true}) : \mathsf{Nat}$, making the addition well-typed.

The meet operation gives us a means to combine different pieces of evidence for a value, which allows evidence to evolve and gain precision as a program runs. However, we also need operations on types, to extract typing information for particular components of a type, such as the domain and codomain of a function. These operations are partial: they produce $?$ when given $?$, produce a result when given a type of the expected shape, and are undefined otherwise. We define these operations in Figure 5.

We note that gradual typing usually begins with a definition of consistency, with precision and meet defined in terms of consistency. Since GTFL is not a contribution of our work, we keep our presentation small by defining operations in terms of meet. τ_1 and τ_2 are consistent if their meet exists, and τ_1 is more precise than τ_2 if $\tau_1 \sqcap \tau_2 = \tau_1$.

$\boxed{\varepsilon \vdash \tau_1 \cong \tau_2}$ *(Type Consistency relative to Evidence)*

$$\textsc{ConsistentEv}\;\frac{\tau_3 \sqcap \tau_1 = \tau_3 \qquad \tau_3 \sqcap \tau_2 = \tau_3}{\{\tau_3\} \vdash \tau_1 \cong \tau_2}$$

$\boxed{\tau_1 \sqcap \tau_2 = \tau_3}$ *(Precision Meet)*

$$\textsc{MeetDynL}\;\frac{}{? \sqcap \tau = \tau} \qquad \textsc{MeetDynR}\;\frac{}{\tau \sqcap ? = \tau} \qquad \textsc{MeetRefl}\;\frac{}{\tau \sqcap \tau = \tau}$$

$$\textsc{MeetFun}\;\frac{\tau_1 \sqcap \tau_1' = \tau_1'' \qquad \tau_2 \sqcap \tau_2' = \tau_2''}{\tau_1 \to \tau_2 \sqcap \tau_1' \to \tau_2' = \tau_1'' \to \tau_2''}$$

$$\textsc{MeetProd}\;\frac{\tau_1 \sqcap \tau_1' = \tau_1'' \qquad \tau_2 \sqcap \tau_2' = \tau_2''}{\tau_1 \times \tau_2 \sqcap \tau_1' \times \tau_2' = \tau_1'' \times \tau_2''}$$

Figure 4: Source Language: Type Consistency and Precision

$$\begin{array}{lcll}
\textbf{dom}\,(\tau_1 \to \tau_1) & = & \tau_1 & \\
\textbf{dom}\,? & = & ? & \\
\textbf{cod}\,(\tau_1 \to \tau_1) & = & \tau_2 & \\
\textbf{cod}\,? & = & ? & \\
\textbf{Proj}_i(\tau_1 \times \tau_2) & = & \tau_i & i \in \{1,2\} \\
\textbf{Proj}_i\,? & = & ? &
\end{array}$$

Figure 5: Source Language: Partial Type Operations

2.4 Runtime Semantics

$\boxed{\mathsf{e}_1 \longrightarrow \mathsf{e}_2}$ *(Reduction Relation on terms)*

$$\textsc{RedIfTrue}\quad \frac{}{\textbf{if true then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2 \longrightarrow \mathsf{e}_1}$$

$$\textsc{RedIfFalse}\quad \frac{}{\textbf{if false then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2 \longrightarrow \mathsf{e}_2}$$

$$\textsc{RedIfEv}\quad \frac{\textbf{if } b \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2 \longrightarrow \mathsf{e}}{\textbf{if } \varepsilon\, b \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2 \longrightarrow \mathsf{e}}$$

$$\textsc{RedApp}\quad \frac{}{(\lambda x.\, \mathsf{e})\ \mathsf{v} \longrightarrow [\mathsf{v}/x]\mathsf{e}}$$

$$\textsc{RedAppEv}\quad \frac{}{(\varepsilon_1\ (\lambda x.\, \mathsf{e}))\ (\varepsilon_2\ \mathsf{r}) \longrightarrow (\textbf{cod}\, \varepsilon_2)\ ([(\textbf{dom}\, \varepsilon_1 \sqcap \varepsilon_2)\ \mathsf{r}/x]\mathsf{e})}$$

$$\textsc{RedAppEvFail}\quad \frac{\textbf{dom}\, \varepsilon_1 \sqcap \varepsilon_2 \textbf{ undefined}}{(\varepsilon_1\ (\lambda x.\, \mathsf{e}))\ (\varepsilon_2\ \mathsf{r}) \longrightarrow \textbf{error}}$$

$$\textsc{RedAppEvPartial}\quad \frac{}{(\varepsilon\ (\lambda x.\, \mathsf{e}_1))\ \mathsf{r} \longrightarrow (\textbf{cod}\, \varepsilon)\ ([(\textbf{dom}\, \varepsilon)\ \mathsf{e}_2/x]\mathsf{e}_1)}$$

$$\textsc{RedAppEvPartialFail}\quad \frac{\textbf{dom}\, \varepsilon \textbf{ undefined}}{(\varepsilon\ (\lambda x.\, \mathsf{e}_1))\ \mathsf{v} \longrightarrow \textbf{error}}$$

$$\textsc{RedPlus}\quad \frac{}{n_1 + n_2 \longrightarrow n_1 + n_2}$$

$$\textsc{RedPlusEvL}\quad \frac{n_1 + \mathsf{v} \longrightarrow \mathsf{e}}{(\varepsilon\ n_1) + \mathsf{v} \longrightarrow \mathsf{e}}$$

$$\textsc{RedPlusEvR}\quad \frac{n_1 + n_2 \longrightarrow \mathsf{e}}{n_1 + \varepsilon\ n_2 \longrightarrow \mathsf{e}}$$

$$\textsc{RedEqT}\quad \frac{}{n \stackrel{?}{=} n \longrightarrow \textbf{true}}$$

$$\textsc{RedEqF}\quad \frac{n_1 \neq n_2}{n_1 \stackrel{?}{=} n_2 \longrightarrow \textbf{false}}$$

$$\textsc{RedEqEvL}\quad \frac{n_1 \stackrel{?}{=} \mathsf{v} \longrightarrow \mathsf{e}}{(\varepsilon\ n_1) \stackrel{?}{=} \mathsf{v} \longrightarrow \mathsf{e}}$$

$$\textsc{RedEqEvR}\quad \frac{n_1 \stackrel{?}{=} n_2 \longrightarrow \mathsf{e}}{n_1 \stackrel{?}{=} \varepsilon\ n_2 \longrightarrow \mathsf{e}}$$

$$\textsc{RedProj}\quad \frac{i \in \{1, 2\}}{\pi_i \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \longrightarrow \mathsf{e}_i}$$

$$\textsc{RedProjEv}\quad \frac{i \in \{1, 2\}}{\pi_i (\varepsilon\ \langle \mathsf{e}_1, \mathsf{e}_2 \rangle) \longrightarrow (\textbf{Proj}_i \varepsilon)\ \mathsf{e}_i}$$

$$\textsc{RedProjFail}\quad \frac{\textbf{Proj}_i \varepsilon \textbf{ undefined}}{\pi_i (\varepsilon\ \langle \mathsf{e}_1, \mathsf{e}_2 \rangle) \longrightarrow \textbf{error}}$$

$$\textsc{RedAscr}\quad \frac{}{\varepsilon_1\ (\varepsilon_2\ \mathsf{r}) \longrightarrow (\varepsilon_1 \sqcap \varepsilon_2)\ \mathsf{r}}$$

$$\textsc{RedAscrFail}\quad \frac{\varepsilon_1 \sqcap \varepsilon_2 \textbf{ undefined}}{\varepsilon_1\ (\varepsilon_2\ \mathsf{r}) \longrightarrow \textbf{error}}$$

$$\textsc{RedContext}\quad \frac{\mathsf{e}_1 \longrightarrow \mathsf{e}_2}{C[\mathsf{e}_1] \longrightarrow C[\mathsf{e}_2]}$$

$$\textsc{RedContextFail}\quad \frac{\mathsf{e} \longrightarrow \textbf{error}}{C[\mathsf{e}] \longrightarrow C[\textbf{error}]}$$

Figure 6: Source Language: Small-Step Operational Semantics

As we saw above, $1 + \{\mathsf{Nat}\}\ (\{\mathsf{Bool}\}\ \textbf{true})$ was assigned a type in our language. But how should such a term behave? Allowing it to result in any integer value would require an arbitrary choice, so the only safe result of such a computation is **error**. Specifically, because values may only contain one piece of top-level evidence, computation fails trying to combine the evidence objects Bool and Nat, since their meet is undefined.

We present the full semantics for GTFL in Figure 6. In general, we have rules which one expects in a static language, plus rules accounting for values with evidence. When we have nested evidence it is combined with RedAscr. When applying a function using the rule RedAppEv, we must first use domain information from the function's evidence to convert the argument to the type the function expects. The result is then ascribed with the codomain information from the function's evidence. These evidence operations mirror those of higher-order contracts [Findler and Felleisen, 2002]. We decompose the evidence for pairs in a similar way for projections in RedProjEv.

For primitive operations, we simply ignore evidence, as any well-typed values must have the appropriate type. Similarly, in RedAppEvPartial, if we apply a function with evidence to a raw value, then we treat the argument as if it had been ascribed evidence ?.

If any of the evidence operations in the above rules are undefined, then the only way to preserve safety is to step to **error**, which is what happens in RedAppEvFail, RedProjFail and RedAscrFail. Context frames are defined in Figure 7, and RedContext allows us to step within any context frame. Similarly, errors are propagated with RedContextFail. The context rules establish a left-to-right, call-by-value evaluation order.

While somewhat complex, basing a gradual language around evidence has several advantages. First, the AGT approach allows us to take a pre-existing static language and introduce gradual types. Second, various properties of the language, such as type safety, hold by construction when using the evidence approach.

$$
\begin{array}{llll}
C & ::= & & \text{Context Frames} \\
 & | & \square\, e & \\
 & | & v\, \square & \\
 & | & (\square, e) & \\
 & | & (v, \square) & \\
 & | & \pi_1\, \square & \\
 & | & \pi_2\, \square & \\
 & | & \varepsilon\, \square & \\
 & | & \square + e_1 & \\
 & | & v + \square & \\
 & | & \square \stackrel{?}{=} e_1 & \\
 & | & v \stackrel{?}{=} \square & \\
 & | & \textbf{if}\, \square\, \textbf{then}\, e_1\, \textbf{else}\, e_2 &
\end{array}
$$

Figure 7: Source Language: Context Frames

3 The Target Language

Our target language, given in Figure 8, is essentially an untyped version of the λ^K calculus presented by Morrisett et al. [1999]. We have distinct syntactic classes for *values* (metavariable *u*), and *terms* (metavariable *t*). Each syntactic form for terms denotes a single operation on a value, and any nested computations must be explicitly represented with the passing of continuations. We do not provide a semantics for the target, but note that it is straightforward, using β-reductions, substitution for **let**, and primitive operations in the usual way.

Notably, our target language is *not* gradual. Because gradual types let us write terms that have no purely static type, we treat our target as untyped typed.

$$
\begin{array}{lll}
u,\ k & ::= & \\
& | & x \\
& | & n \\
& | & b \\
& | & \mathbf{fix}\ x\ u \\
& | & \lambda x_1 \ldots x_i.\ t \\
& | & \langle u_1, u_2 \rangle
\end{array}
$$

$$
\begin{array}{lll}
d & ::= & \\
& | & x := u \\
& | & x := \pi_1 u \\
& | & x := \pi_2 u \\
& | & x := u_1 + u_2 \\
& | & x := u_1 \stackrel{?}{=} u_2
\end{array}
$$

$$
\begin{array}{lll}
t & ::= & \\
& | & \mathsf{v} \\
& | & \mathbf{let}\ d\ \mathbf{in}\ t \\
& | & u(arg) \\
& | & \mathbf{if}\ u\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\
& | & \mathbf{halt}\ [u] \\
& | & \mathbf{error}
\end{array}
$$

Figure 8: Target Language: Syntax

4 The Translation

With our source and target languages defined, we can specify a translation between the two. The key idea is that the evidence information in the source must be explicitly represented using nested pairs (effectively untyped trees) and integer tags. While operations on evidence, such as meet, were taken as atomic in the source, we provide explicit target implementations for these, and to ensure safety, sequence these operations before the operations on values are performed.

4.1 Translating Evidence

In subsection 4.1, we translate each evidence object into a pair. The first element of the pair is a tag, denoting the root type-constructor for the type. We assume we have defined distinct integer constants $\mathrm{NAT}, \mathrm{BOOL}, \mathrm{ARROW}, and \mathrm{PRODUCT}$. For the simple types Bool, Nat and $?$, we simply place a dummy value as the second pair element, but for compound function and product types, we insert a pair containing the (recursively computed) representation of the sub-components.

TODO

$\boxed{\mathcal{T}[\![\varepsilon]\!] = u}$ *(CPS Representation of Runtime Evidence)*

$$\frac{\textsc{EvTransformBool}}{\mathcal{T}[\![\{\mathsf{Bool}\}]\!] = \langle \mathrm{BOOL}, 0\rangle} \qquad \frac{\textsc{EvTransformNat}}{\mathcal{T}[\![\{\mathsf{Nat}\}]\!] = \langle \mathrm{NAT}, 0\rangle} \qquad \frac{\textsc{EvTransformDyn}}{\mathcal{T}[\![\{?\}]\!] = \langle \mathrm{DYN}, 0\rangle}$$

$$\frac{\textsc{EvTransformArr}}{\mathcal{T}[\![\{\tau_1 \rightarrow \tau_2\}]\!] = \langle \mathrm{ARROW}, \langle \mathcal{T}[\![\{\tau_1\}]\!], \mathcal{T}[\![\{\tau_2\}]\!]\rangle\rangle}$$

$$\frac{\textsc{EvTransformProd}}{\mathcal{T}[\![\{\tau_1 \rightarrow \tau_2\}]\!] = \langle \mathrm{PRODUCT}, \langle \mathcal{T}[\![\{\tau_1\}]\!], \mathcal{T}[\![\{\tau_2\}]\!]\rangle\rangle}$$

Translation: Evidence

4.1.1 Helper Functions

With our evidence represented as tuples with integer tags, we must represent the partial functions on types in our target language. The implementation is given in Figure 9. Doing this is straightforward: if one argument is $?$, then we return the other argument. Otherwise, we check if we have simple or complex types. For simple types, either $\mathsf{Bool} \sqcap \mathsf{Bool} = \mathsf{Bool}$, $\mathsf{Nat} \sqcap \mathsf{Nat} = \mathsf{Nat}$. For complex types, we check that the tags agree, then recursively compute the meets of the sub-components. If neither argument is $?$, and there is a tag mismatch, then we must raise an exception, retuning the **error** continuation.

For the partial functions decomposing types, we first check if the input is $?$, in which case we return $?$. Otherwise, we check the tag, and if it is correct, we return the relevant sub-component of the type. In all other cases, we throw an error. We give an example implementation for **dom** in Figure 10: either we are given $?$ and return $?$, we are given $\tau_1 \rightarrow \tau_2$ and we return τ_1, or we raise an exception. We omit **cod**, $\mathbf{proj}_1$ and $\mathbf{proj}_2$, but they are implemented similarly.

$$
\begin{aligned}
\text{MEET} \quad = & \,\mathbf{fix}\,\text{self}\,\lambda \text{ty1 ty2}\, c.\, \mathbf{let}\,\text{tag1} := \pi_1 \text{ty1}\,\mathbf{in}\,\mathbf{let}\,\text{sub1} := \pi_2 \text{ty1}\,\mathbf{in}\,\mathbf{let}\,\text{isDyn1} := \text{tag1} \stackrel{?}{=} \text{DYN}\,\mathbf{in} \\
& \mathbf{if}\,\text{isDyn1}\,\mathbf{then}\, c(\text{ty2})\,\mathbf{else} \\
& \mathbf{let}\,\text{tag2} := \pi_1 \text{ty2}\,\mathbf{in}\,\mathbf{let}\,\text{sub2} := \pi_2 \text{ty2}\,\mathbf{in}\,\mathbf{let}\,\text{isDyn2} := \text{tag2} \stackrel{?}{=} \text{DYN}\,\mathbf{in} \\
& \mathbf{if}\,\text{isDyn2}\,\mathbf{then}\, c(\text{ty1})\,\mathbf{else} \\
& \mathbf{let}\,\text{isNat1} := \text{tag1} \stackrel{?}{=} \text{NAT}\,\mathbf{in}\,\mathbf{let}\,\text{isNat2} := \text{tag2} \stackrel{?}{=} \text{NAT}\,\mathbf{in} \\
& \mathbf{if}\,\text{isNat1}\,\mathbf{then}\,(\mathbf{if}\,\text{isNat2}\,\mathbf{then}\,k(\text{NAT})\,\mathbf{else}\,\mathbf{error})\,\mathbf{else} \\
& \mathbf{let}\,\text{isBool1} := \text{tag1} \stackrel{?}{=} \text{BOOL}\,\mathbf{in}\,\mathbf{let}\,\text{isBool2} := \text{tag2} \stackrel{?}{=} \text{BOOL}\,\mathbf{in} \\
& \mathbf{if}\,\text{isBool1}\,\mathbf{then}\,(\mathbf{if}\,\text{isBool2}\,\mathbf{then}\,k(\text{BOOL})\,\mathbf{else}\,\mathbf{error})\,\mathbf{else} \\
& \mathbf{let}\,\text{isArrow1} := \text{tag1} \stackrel{?}{=} \text{ARROW}\,\mathbf{in}\,\mathbf{let}\,\text{isArrow2} := \text{tag2} \stackrel{?}{=} \text{ARROW}\,\mathbf{in} \\
& \mathbf{if}\,\text{isArrow1}\,\mathbf{then} \\
& \quad \mathbf{let}\,\text{dom1} := \pi_1 \text{sub1}\,\mathbf{in}\,\mathbf{let}\,\text{cod1} := \pi_2 \text{sub1}\,\mathbf{in} \\
& \quad \mathbf{if}\,\text{isArrow2}\,\mathbf{then} \\
& \quad\quad \mathbf{let}\,\text{dom2} := \pi_1 \text{sub2}\,\mathbf{in}\,\mathbf{let}\,\text{cod2} := \pi_2 \text{sub2}\,\mathbf{in} \\
& \quad\quad \text{self}(\text{dom1}, \text{dom2}, (\lambda \text{meet1}.\, \text{self}(\text{cod1}, \text{cod2}, (\lambda \text{meet2}.\, k(\langle \text{ARROW}, \langle \text{meet1}, \text{meet2} \rangle \rangle)))))) \\
& \quad \mathbf{else}\,\mathbf{error} \\
& \mathbf{let}\,\text{isProduct1} := \text{tag1} \stackrel{?}{=} \text{PRODUCT}\,\mathbf{in}\,\mathbf{let}\,\text{isProduct2} := \text{tag2} \stackrel{?}{=} \text{PRODUCT}\,\mathbf{in} \\
& \mathbf{if}\,\text{isProduct1}\,\mathbf{then} \\
& \quad \mathbf{let}\,\text{lhs1} := \pi_1 \text{sub1}\,\mathbf{in}\,\mathbf{let}\,\text{rhs1} := \pi_2 \text{sub1}\,\mathbf{in} \\
& \quad \mathbf{if}\,\text{isProduct2}\,\mathbf{then} \\
& \quad\quad \mathbf{let}\,\text{lhs2} := \pi_1 \text{sub2}\,\mathbf{in}\,\mathbf{let}\,\text{rhs2} := \pi_2 \text{sub2}\,\mathbf{in} \\
& \quad\quad \text{self}(\text{lhs1}, \text{lhs2}, (\lambda \text{meet1}.\, \text{self}(\text{rhs1}, \text{rhs2}, (\lambda \text{meet2}.\, k(\langle \text{PRODUCT}, \langle \text{meet1}, \text{meet2} \rangle \rangle)))))) \\
& \quad \mathbf{else}\,\mathbf{error} \\
& \mathbf{else}\,\mathbf{error}
\end{aligned}
$$

Figure 9: CPS implementation of meet

$$
\begin{aligned}
\mathrm{DOM} \quad = & \lambda \mathrm{ty}\, c.\, \textbf{let}\, \mathrm{tag} := \pi_1 \mathrm{ty1}\, \textbf{in}\, \textbf{let}\, \mathrm{sub} := \pi_2 \mathrm{ty1}\, \textbf{in}\, \textbf{let}\, \mathrm{isDyn} := \mathrm{tag} \stackrel{?}{=} \mathrm{DYN}\, \textbf{in} \\
& \textbf{if}\ \mathrm{isDyn}\ \textbf{then}\ c(\langle \mathrm{DYN}, 0\rangle)\ \textbf{else} \\
& \textbf{let}\, \mathrm{isArrow} := \mathrm{tag} \stackrel{?}{=} \mathrm{ARROW}\, \textbf{in} \\
& \textbf{if}\ \mathrm{isArrow}\ \textbf{then}\ (\textbf{let}\, \mathrm{ret} := \pi_1 \mathrm{sub}\, \textbf{in}\, k(\mathrm{ret}))\ \textbf{else error}
\end{aligned}
$$

Figure 10: CPS implementation of domain

4.2 Transforming Terms

$\boxed{\mathcal{E}[\![\mathrm{e}]\!]k = t}$ *(CPS Translation of Expressions)*

$$\textsc{TransformVar}\quad \frac{}{\mathcal{E}[\![x]\!]k = k(x)} \qquad \textsc{TransformBool}\quad \frac{}{\mathcal{E}[\![b]\!]k = k(\langle \mathrm{DYN}, b\rangle)} \qquad \textsc{TransformNum}\quad \frac{}{\mathcal{E}[\![n]\!]k = k(\langle \mathrm{DYN}, n\rangle)}$$

$$\textsc{TransformFun}\quad \frac{\mathcal{E}[\![\mathrm{e}]\!]c = t}{\mathcal{E}[\![(\lambda x.\,\mathrm{e})]\!]k = k(\langle \mathrm{DYN}, \lambda x\, c.\, t\rangle)}$$

$$\textsc{TransformApp}\quad \frac{\begin{array}{c} k_1 := (\lambda x_2.\,\mathbf{let}\ y_1 := \pi_1 x_1\ \mathbf{in}\ \mathbf{let}\ z_1 := \pi_2 x_1\ \mathbf{in}\ \mathbf{let}\ y_2 := \pi_1 x_2\ \mathbf{in}\ \mathbf{let}\ z_2 := \pi_2 x_2\ \mathbf{in}\ t_1) \\ t_1 := \mathrm{DOM}(y_1, \lambda y_1'.\,\mathrm{COD}(y_1, \lambda y_1''.\,\mathrm{MEET}(y_1', y_2, (\lambda y_3.\, z_1(\langle y_3, z_2\rangle, (\lambda z_3.\, t_2)))))) \\ t_2 := \mathbf{let}\ z_3' := \pi_1 z_3\ \mathbf{in}\ \mathbf{let}\ z_3'' := \pi_2 z_3\ \mathbf{in}\ \mathrm{MEET}(y_1'', z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle))) \\ \mathcal{E}[\![\mathrm{e}_2]\!]k_1 = t' \\ \mathcal{E}[\![\mathrm{e}_1]\!](\lambda x_1.\, t') = t \end{array}}{\mathcal{E}[\![\mathrm{e}_1\ \mathrm{e}_2]\!]k = t}$$

$$\textsc{TransformPlus}\quad \frac{\begin{array}{c} k_1 := (\lambda x_2.\,\mathbf{let}\ z_1 := \pi_2 x_1\ \mathbf{in}\ \mathbf{let}\ z_2 := \pi_2 x_2\ \mathbf{in}\ \mathbf{let}\ z_3 := z_1 + z_2\ \mathbf{in}\ k(z_3)) \\ \mathcal{E}[\![\mathrm{e}_2]\!]k_1 = t' \\ \mathcal{E}[\![\mathrm{e}_1]\!](\lambda x_1.\, t') = t \end{array}}{\mathcal{E}[\![\mathrm{e}_1 + \mathrm{e}_2]\!]k = t}$$

$$\textsc{TransformEq}\quad \frac{\begin{array}{c} k_1 := (\lambda x_2.\,\mathbf{let}\ z_1 := \pi_2 x_1\ \mathbf{in}\ \mathbf{let}\ z_2 := \pi_2 x_2\ \mathbf{in}\ \mathbf{let}\ z_3 := z_1 \stackrel{?}{=} z_2\ \mathbf{in}\ k(z_3)) \\ \mathcal{E}[\![\mathrm{e}_2]\!]k_1 = t' \\ \mathcal{E}[\![\mathrm{e}_1]\!](\lambda x_1.\, t') = t \end{array}}{\mathcal{E}[\![\mathrm{e}_1 + \mathrm{e}_2]\!]k = t}$$

$$\textsc{TransformPair}\quad \frac{\begin{array}{c} \mathcal{E}[\![\mathrm{e}_2]\!](\lambda x_2.\, k(\langle \mathrm{DYN}, \langle x_1, x_2\rangle\rangle)) = t' \\ \mathcal{E}[\![\mathrm{e}_1]\!](\lambda x_1.\, t') = t \end{array}}{\mathcal{E}[\![\langle \mathrm{e}_1, \mathrm{e}_2\rangle]\!]k = t} \qquad \textsc{TransformIf}\quad \frac{\begin{array}{c} \mathcal{E}[\![\mathrm{e}_1]\!]k = t_1 \\ \mathcal{E}[\![\mathrm{e}_2]\!]k = t_2 \\ \mathcal{E}[\![\mathrm{e}_0]\!](\lambda x_0.\,\mathbf{let}\ x := \pi_2 x_0\ \mathbf{in}\ \mathbf{if}\ x\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2) = t \end{array}}{\mathcal{E}[\![\mathbf{if}\ \mathrm{e}_0\ \mathbf{then}\ \mathrm{e}_1\ \mathbf{else}\ \mathrm{e}_2]\!]k = t}$$

$$\textsc{TransformProj}\quad \frac{\begin{array}{c} \mathcal{E}[\![\mathrm{e}]\!](\lambda x.\,\mathbf{let}\ y_1 := \pi_1 x\ \mathbf{in}\ \mathbf{let}\ y_2 := \pi_2 x\ \mathbf{in}\ \mathrm{PROD}_i(y_1, k')) = t \\ k' := (\lambda z_1.\,\mathbf{let}\ z_2 := \pi_i\ y\ \mathbf{in}\ \mathbf{let}\ z_{21} := \pi_1 z_2\ \mathbf{in}\ \mathbf{let}\ z_{22} := \pi_2 z_2\ \mathbf{in}\ \mathrm{MEET}(z_1, z_{21}, (\lambda z_1'.\, k(\langle z_1', z_{22}\rangle)))) \end{array}}{\mathcal{E}[\![\pi_i \mathrm{e}]\!]k = t}$$

$$\textsc{TransformEv}\quad \frac{\mathcal{E}[\![\mathrm{e}]\!](\lambda x.\,\mathbf{let}\ x_1 := \pi_1 x\ \mathbf{in}\ \mathbf{let}\ x_2 := \pi_2 x\ \mathbf{in}\ \mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], x_1, (\lambda y.\, k(\langle y, x_2\rangle)))) = t}{\mathcal{E}[\![\varepsilon\ \mathrm{e}]\!]k = t}$$

$$\textsc{TransformErr}\quad \frac{}{\mathcal{E}[\![\mathbf{error}]\!]k = \mathbf{error}}$$

Translation: Terms

5 Whole Program Correctness

5.1 Value Transformations

Since we use small-step semantics, our reduction rules specify how to perform individual operations on values, with context rules for performing nested computations. However, our translation does not distinguish between, for example, a pair containing values and a pair containing reducible expressions. In order to reason about the relationship between our reductions and our translation, we define a special translation for values, which will aid in reasoning about how our translation behaves when given values as input.

$\boxed{\mathcal{V}[\![\mathsf{v}]\!] = u}$ *(CPS Translation of Closed Values)*

$$\textsc{ValTransformBool}\quad \frac{}{\mathcal{V}[\![b]\!] = \langle \mathrm{DYN}, b\rangle}$$

$$\textsc{ValTransformNum}\quad \frac{}{\mathcal{V}[\![n]\!] = \langle \mathrm{DYN}, n\rangle}$$

$$\textsc{ValTransformFun}\quad \frac{\mathcal{E}[\![\mathsf{e}]\!]c = t}{\mathcal{V}[\![\lambda x.\, \mathsf{e}]\!] = \langle \mathrm{DYN}, \lambda x\, c.\, t\rangle}$$

$$\textsc{ValTransformPair}\quad \frac{}{\mathcal{V}[\![\langle \mathsf{v}_1, \mathsf{v}_2\rangle]\!] = \langle \mathrm{DYN}, \langle \mathcal{V}[\![\mathsf{v}_1]\!], \mathcal{V}[\![\mathsf{v}_2]\!]\rangle\rangle}$$

$$\textsc{ValTransformEv}\quad \frac{\mathcal{V}[\![\mathsf{r}]\!] = \langle \mathrm{DYN}, u\rangle}{\mathcal{V}[\![\varepsilon\ \mathsf{r}]\!] = \langle \mathcal{T}[\![\varepsilon]\!], u\rangle}$$

Translation: Values

5.2 Helper Lemmas

We first show that our translation of evidence preserves the semantics of operations on evidence.

Lemma 5.1 (Correctness of Evidence Translation)**.** *Consider evidence $\varepsilon, \varepsilon'$. Then, for any k:*

- $\mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], \mathcal{T}[\![\varepsilon']\!], k) \longrightarrow^* k(\mathcal{T}[\![\varepsilon \sqcap \varepsilon']\!])$ *if $\varepsilon \sqcap \varepsilon'$ is defined.*
- *If $\varepsilon \sqcap \varepsilon'$ is undefined, then* $\mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], \mathcal{T}[\![\varepsilon']\!]) \longrightarrow^* \mathbf{error}$.

The same property holds for $\mathbf{dom}\,\varepsilon$, $\mathbf{cod}\,\varepsilon$, *and* $\mathbf{Proj}_i\varepsilon$.

Next, we show that our value translation always produces pairs. This is essential, since our source semantics can distinguish between terms with and without evidence. However, since there is no explicit evidence in our target, we represent all source values as evidence-value pairs in the target, with unascribed values simply having the evidence DYN.

Lemma 5.2 (Canonical Forms for Translated Values)**.** *For an irreducible* v, $\mathcal{V}[\![\mathsf{v}]\!] = \langle \mathcal{T}[\![\varepsilon]\!], u\rangle$ *for some evidence ε and CPS-value u. Moreover, if v is a raw irreducible, then $\varepsilon = \{?\}$.*

Proof. By inversion on the definition of $\mathcal{V}[\![\mathsf{v}]\!]$. □

Next, we show that our value translation matches our term translation, after perhaps performing some computation. This lemma is crucial for connecting the behaviour of small-step reduction rules, which operate primarily on values, to the behaviour of the translations of these values.

Lemma 5.3 (Value and Expression Translations Match). *Let* v *be an irreducible term. Then, for any* k, v, $\mathcal{E}[\![\mathsf{v}]\!]k \longrightarrow^{*} k(\mathcal{V}[\![\mathsf{v}]\!])$.

Proof. By induction on v.

- Case $\mathsf{v} = b$, $\mathsf{v} = n$, or $\mathsf{v} = \lambda x.\, \mathsf{e}$: trivial.
- Case $\mathsf{v} = \langle \mathsf{v}_1, \mathsf{v}_2 \rangle$. So $\mathcal{E}[\![\langle \mathsf{v}_1, \mathsf{v}_2 \rangle]\!]k = \mathcal{E}[\![\mathsf{v}_1]\!](\lambda x_1.\, \mathcal{E}[\![\mathsf{v}_2]\!](\lambda x_2.\, k(\langle \mathrm{DYN}, \langle x_1, x_2 \rangle \rangle)))$, which, by our hypothesis, reduces to $t_1 \longrightarrow^{*} (\lambda x_1.\, (\lambda x_2.\, k(\langle \mathrm{DYN}, \langle x_1, x_2 \rangle \rangle))(\mathcal{V}[\![\mathsf{v}_2]\!]))(\mathcal{V}[\![\mathsf{v}_1]\!])$, which we can then reduce to $k(\langle \mathrm{DYN}, \langle \mathcal{V}[\![\mathsf{v}_1]\!], \mathcal{V}[\![\mathsf{v}_2]\!] \rangle \rangle)$.
- Case $\mathsf{v} = \varepsilon\, \mathsf{r}$. Since all raw values are themselves irreducible, our inductive hypothesis gives that $\mathcal{E}[\![\mathsf{r}]\!](\lambda x.\, \mathbf{let}\, x_1 := \pi_1 x\, \mathbf{in}\, \mathbf{let}\, x_2 := \pi_2 x\, \mathbf{in}\, \mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], x_1, (\lambda y.\, k(\langle y, x_2 \rangle))))$ steps to $(\lambda x.\, \mathbf{let}\, x_1 := \pi_1 x\, \mathbf{in}\, \mathbf{let}\, x_2 := \pi_2 x\, \mathbf{in}\, \mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], x_1, (\lambda y.\, k(\langle y, x_2 \rangle))))(\mathcal{V}[\![\mathsf{r}]\!])$. By Lemma 5.2, $\mathcal{V}[\![\mathsf{r}]\!]$ is of the form $\langle \mathrm{DYN}, u \rangle$ for some u. So we can then β-reduce and apply the let-substitutions to reach $\mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], \mathrm{DYN}, u)$. By Lemma 5.1, this steps to $\langle \mathcal{T}[\![\varepsilon]\!], u \rangle$. By the rule TransformEv, this means that $\mathcal{V}[\![\varepsilon\, \mathsf{r}]\!]$ also steps to this value.

□

Finally, we show that our translation preserves substitution, which is needed to show the correctness of our translation of functions.

Lemma 5.4 (Translation Commutes With Substitution). $\mathcal{E}[\![[\mathsf{v}/x]\mathsf{e}]\!]([\mathcal{V}[\![\mathsf{v}]\!]/x]k) \longrightarrow^{*} [\mathcal{V}[\![\mathsf{v}]\!]/x]\mathcal{E}[\![\mathsf{e}]\!]k$.

Proof. Follows from straightforward induction on e, combined with Lemma 5.3 for the case where $\mathsf{e} = x$. □

5.3 Main Result

These lemmas give us the tools we need to prove our main result. We show that the translation takes a source term to a target term that is *equivalent* to the result of reduction Essentially, we are saying that if $\mathsf{e}_1 \longrightarrow \mathsf{e}_2$, then e_1 and e_2 translate to target terms that will eventually step to some common term.

Theorem 5.1 (Weak Simulation). *If* $\mathsf{e}_1 \longrightarrow \mathsf{e}_2$, *then for all* k, $\mathcal{E}[\![\mathsf{e}_1]\!]k \equiv \mathcal{E}[\![\mathsf{e}_2]\!]k$.

Proof. We perform induction on the derivation tree of $\mathsf{e}_1 \longrightarrow \mathsf{e}_2$.

- RedIfTrue: then $e_1 = \textbf{if true then } e_2 \textbf{ else } e_3$. The translation $\mathcal{E}[\![\textbf{true}]\!]k' = k'(\langle \mathrm{DYN}, \textbf{true}\rangle)$ for any k', so $\mathcal{E}[\![\textbf{if true then } e_2 \textbf{ else } e_3]\!]k$ is $(\lambda x_0.\, \textbf{let } x := \pi_2 x_0 \textbf{ in if } x \textbf{ then } (\mathcal{E}[\![e_2]\!]k) \textbf{ else } (\mathcal{E}[\![e_3]\!]k))(\langle \mathrm{DYN}, \textbf{true}\rangle)$. We can β-reduce to get $\textbf{let } x := \pi_2\langle \mathrm{DYN}, \textbf{true}\rangle \textbf{ in if } x \textbf{ then } \mathcal{E}[\![e_2]\!]k \textbf{ else } \mathcal{E}[\![e_3]\!]k$, and we can substitute **true** for x and reduce the **if** to get $\mathcal{E}[\![e_2]\!]k$.

- RedIfFalse: symmetric to RedIfTrue

- RedIfEv: $e_1 = \textbf{if } \varepsilon\, b \textbf{ then } e_2' \textbf{ else } e_3'$. We know that $\mathcal{E}[\![b]\!]k' = k'(\langle \mathrm{DYN}, b\rangle)$, so $\mathcal{E}[\![\varepsilon\, b]\!]k'' = (\lambda x.\, \textbf{let } x_1 := \pi_1 x \textbf{ in let } x_2 := \pi_2 x \textbf{ in } \mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], x_1, (\lambda y.\, k''(\langle y, x_2\rangle)))))(\langle \mathrm{DYN}, b\rangle)$. We can β-reduce, and substitute with the let-expressions, to get $(\lambda x.\, \mathrm{MEET}(\mathcal{T}[\![\varepsilon]\!], \mathrm{DYN}, (\lambda y.\, k''(\langle y, b\rangle))))$. However, $\varepsilon \sqcap \{?\} = \{?\}$, so by Lemma 5.1 this steps to $k''(\langle \mathcal{T}[\![\varepsilon]\!], b\rangle)$. Since the translation of **if** ignores any evidence in the condition, we can use the same reasoning from RedIfTrue to show that it steps to e_2 if b is true, and e_3 if b is false.

- RedApp: then $e_1 = (\lambda x.\, e')\, v$ and $e_2 = [v/x]e'$. We assume our terms follow variable convention so that x is not free in k.

  Let $\langle \mathcal{T}[\![\varepsilon]\!], u\rangle = \mathcal{V}[\![v]\!]$ (by Lemma 5.2). If we apply Lemma 5.3, we can see that $\mathcal{E}[\![(\lambda x.\, e')\, v]\!]k$ steps to $(\lambda x_1\, x_2.\, \textbf{let } y_1 := \pi_1 x_1 \textbf{ in } \ldots)(\langle \mathrm{DYN}, (\lambda x\, c.\, \mathcal{E}[\![e']\!]c)\rangle, \langle \mathcal{T}[\![\varepsilon]\!], u\rangle)$. We can β-reduce and apply the let-substitutions to then step to $\mathrm{DOM}(\mathrm{DYN}, \lambda y_1'.\, \mathrm{COD}(\mathrm{DYN}, \lambda y_1''.\, \mathrm{MEET}(y_1', \mathcal{T}[\![\varepsilon]\!], (\lambda y_3.\, (\lambda x\, c.\, \mathcal{E}[\![e']\!]c)(\langle y_3, u\rangle, (\lambda z_3.\, \textbf{let } z_3' := \pi_1 z_3 \textbf{ in let } z_3'' := \pi_2 z_3 \textbf{ in } \mathrm{MEET}(y_1'', z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle)))))))))$. By applying Lemma 5.1 for DOM, COD and MEET of $?$ respectively, we can step to $(\lambda x\, c.\, \mathcal{E}[\![e']\!]c)(\langle \mathcal{T}[\![\varepsilon]\!], u\rangle, (\lambda z_3.\, \textbf{let } z_3' := \pi_1 z_3 \textbf{ in let } z_3'' := \pi_2 z_3 \textbf{ in } \mathrm{MEET}(\mathrm{DYN}, z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle)))))$. This then β-reduces to $[\langle \mathcal{T}[\![\varepsilon]\!], u\rangle/x]\mathcal{E}[\![e']\!](\lambda z_3.\, \textbf{let } z_3' := \pi_1 z_3 \textbf{ in let } z_3'' := \pi_2 z_3 \textbf{ in } \mathrm{MEET}(\mathrm{DYN}, z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle))))$. But, then, by Lemma 5.1 and η-equivalence, this is equivalent to $[\langle \mathcal{T}[\![\varepsilon]\!], u\rangle/x]\mathcal{E}[\![e']\!]k$. But we know that this is $[\mathcal{V}[\![v]\!]/x]\mathcal{E}[\![e']\!]k$ Finally, our variable convention and Lemma 5.4 give us that this is equivalent to $\mathcal{E}[\![[v/x]e']\!]k$.

- RedAppEv: then $e_1 = \varepsilon_1\, (\lambda x.\, e')\, \varepsilon_2\, v$ and $e_2 = \textbf{cod}\, \varepsilon_1\, ([(\textbf{dom}\, \varepsilon_1 \sqcap \varepsilon_2)\, v/x]e')$. Let $\langle \mathcal{T}[\![\varepsilon_2]\!], u\rangle = \mathcal{V}[\![v]\!]$ (by Lemma 5.2). If we apply Lemma 5.3, we can see that $\mathcal{E}[\![\varepsilon_1\, (\lambda x.\, e')\, \varepsilon_2\, v]\!]k$ steps to $(\lambda x_1\, x_2.\, \textbf{let } y_1 := \pi_1 x_1 \textbf{ in } \ldots)(\langle \mathcal{T}[\![\varepsilon_1]\!], (\lambda x\, c.\, \mathcal{E}[\![e']\!]c)\rangle, \langle \mathcal{T}[\![\varepsilon_2]\!], u\rangle)$. We can β-reduce and apply the let-substitutions to then step to $\mathrm{DOM}(\mathcal{T}[\![\varepsilon_1]\!], \lambda y_1'.\, \mathrm{COD}(\mathcal{T}[\![\varepsilon_1]\!], \lambda y_1''.\, \mathrm{MEET}(y_1', \mathcal{T}[\![\varepsilon_2]\!], (\lambda y_3.\, (\lambda x\, c.\, \mathcal{E}[\![e']\!]c)(\langle y_3, u\rangle, (\lambda z_3.\, \textbf{let } z_3' := \pi_1 z_3 \textbf{ in let } z_3'' := \pi_2 z_3 \textbf{ in } \mathrm{MEET}(y_1'', z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle)))))))))$. By applying Lemma 5.1 for DOM, COD and MEET of $?$ respectively, we can step to $(\lambda x\, c.\, \mathcal{E}[\![e']\!]c)(\langle \mathcal{T}[\![\textbf{dom}\, \varepsilon_1 \sqcap \varepsilon_2]\!], u\rangle, (\lambda z_3.\, \textbf{let } z_3' := \pi_1 z_3 \textbf{ in let } z_3'' := \pi_2 z_3 \textbf{ in }$ $\mathrm{MEET}(\mathcal{T}[\![\textbf{cod}\, \varepsilon_1]\!], z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle)))))$. This then β-reduces to

$[\langle \mathcal{T}[\![\mathbf{dom}\,\varepsilon_1 \sqcap \varepsilon_2]\!], u\rangle / x]\mathcal{E}[\![\mathsf{e}']\!](\lambda z_3.\, \mathbf{let}\, z_3' := \pi_1 z_3 \,\mathbf{in}\, \mathbf{let}\, z_3'' := \pi_2 z_3 \,\mathbf{in}$ $\mathrm{MEET}(\mathcal{T}[\![\mathbf{cod}\,\varepsilon_1]\!], z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle))))$.
But, by the rule TransformEv, this is α-equivalent to $\mathcal{E}[\![\mathbf{cod}\,\varepsilon_1\,([(\mathbf{dom}\,\varepsilon_1 \sqcap \varepsilon_2)\,\mathsf{v}/x]\mathsf{e}')]\!]k$, giving us our result.

- RedAppEvFail: then $\mathsf{e}_1 = \varepsilon_1\,(\lambda x.\, \mathsf{e}')\,\varepsilon_2\,\mathsf{v}$ and $\mathsf{e}_2 = \mathbf{error}$. Let $\langle \mathcal{T}[\![\varepsilon_2]\!], u\rangle = \mathcal{V}[\![\mathsf{v}]\!]$ (by Lemma 5.2). If we apply Lemma 5.3, we can see that $\mathcal{E}[\![\varepsilon_1\,(\lambda x.\, \mathsf{e}')\,\varepsilon_2\,\mathsf{v}]\!]k$ steps to $(\lambda x_1\, x_2.\, \mathbf{let}\, y_1 := \pi_1 x_1 \,\mathbf{in}\, \ldots)(\langle \mathcal{T}[\![\varepsilon_1]\!], (\lambda x\, c.\, \mathcal{E}[\![\mathsf{e}']\!]c)\rangle, \langle \mathcal{T}[\![\varepsilon_2]\!], u\rangle)$. We can β-reduce and apply the let-substitutions to then step to $\mathrm{DOM}(\mathcal{T}[\![\varepsilon_1]\!], \lambda y_1'.\, \mathrm{COD}(\mathcal{T}[\![\varepsilon_1]\!], \lambda y_1''.\, \mathrm{MEET}(y_1', \mathcal{T}[\![\varepsilon_2]\!], (\lambda y_3.\, (\lambda x\, c.\, \mathcal{E}[\![\mathsf{e}']\!]c)(\langle y_3, u\rangle, (\lambda z_3.\, \mathbf{let}\, z_3' := \pi_1 z_3 \,\mathbf{in}\, \mathbf{let}\, z_3'' := \pi_2 z_3 \,\mathbf{in}\, \mathrm{MEET}(y_1'', z_3', (\lambda z_4.\, k(\langle z_4, z_3''\rangle)))))))))$. By applying Lemma 5.1 for MEET with our premise that $\mathbf{dom}\,\varepsilon_1 \sqcap \varepsilon_2$ **undefined** we can step to **error**.

- RedAppEvPartial: the same reasoning as RedAppEv, except that by Lemma 5.3, we know that the argument's translation is annotated with $?$.

- RedAppEvFailPartial: the same reasoning as RedfAppEvFail, except for DOM instead of MEET.

- RedPlus, RedEqT, RedEqF, RedProj: trivial.

- RedPlusEvL, RedPlusEvR, RedEqEvL, RedEqEvR: follows from our induction hypothesis, combined with the fact that the TransformPlus and TransformEq both ignore evidence.

- RedProj Then $\mathsf{e}_1 = \pi_i\langle \mathsf{v}_1,\mathsf{v}_2\rangle$ and $\mathsf{e}_2 = \mathsf{v}_i$. Then combining TransformProj with Lemma 5.2 and Lemma 5.3, we have $\mathcal{E}[\![\pi_i\langle \mathsf{v}_1,\mathsf{v}_2\rangle]\!]k$ steps to $(\lambda x.\, \mathbf{let}\, y_1 := \pi_1 x \,\mathbf{in}\, \ldots)(\langle \mathrm{DYN}, \langle \mathcal{V}[\![\mathsf{v}_1]\!], \mathcal{V}[\![\mathsf{v}_2]\!]\rangle\rangle)$. If we β-reduce and substitute for the let-expressions, we get $\mathrm{PROD}_i(\mathrm{DYN}, (\lambda z_1.\ldots))$. We apply Lemma 5.1 and let-substitution to step to $\mathrm{MEET}(\mathrm{DYN}, u_1, (\lambda z_1'.\, k(\langle z_1', u_2\rangle)))$, where $\mathcal{V}[\![\mathsf{v}_i]\!] = \langle u_1, u_2\rangle$ by Lemma 5.2. By Lemma 5.1 this steps to $k(\langle u_1, u_2\rangle)$ which we can also step to from $\mathcal{E}[\![\mathsf{v}_i]\!]k$ by Lemma 5.3.

- RedProjEv Then $\mathsf{e}_1 = \pi_i(\varepsilon\,\langle \mathsf{v}_1,\mathsf{v}_2\rangle)$ and $\mathsf{e}_2 = (\mathbf{Proj}_i\varepsilon)\,\mathsf{v}_i$. Then combining TransformProj with Lemma 5.2 and Lemma 5.3, we have $\mathcal{E}[\![\pi_i\langle \mathsf{v}_1,\mathsf{v}_2\rangle]\!]k$ steps to $(\lambda x.\, \mathbf{let}\, y_1 := \pi_1 x \,\mathbf{in}\, \ldots)(\langle \mathcal{T}[\![\varepsilon]\!], \langle \mathcal{V}[\![\mathsf{v}_1]\!], \mathcal{V}[\![\mathsf{v}_2]\!]\rangle\rangle)$. If we β-reduce, and substitute for the let-expressions, we get $\mathrm{PROD}_i(\mathcal{T}[\![\varepsilon]\!], (\lambda z_1.\ldots))$. We apply Lemma 5.1 and let-substitution to step to $\mathrm{MEET}(\mathcal{T}[\![\mathbf{Proj}_i\varepsilon]\!], u_1, (\lambda z_1'.\, k(\langle z_1', u_2\rangle)))$, where $\mathcal{V}[\![\mathsf{v}_i]\!] = \langle u_1, u_2\rangle$ by Lemma 5.2.

  Looking now at e_2, we can apply TransformEv and Lemma 5.2, then β-reduce and substitute to see that $\mathcal{E}[\![(\mathbf{Proj}_i\varepsilon)\,\mathsf{v}_i]\!]k$ also steps to $\mathrm{MEET}(\mathcal{T}[\![\mathbf{Proj}_i\varepsilon]\!], u_1, (\lambda z_1'.\, k(\langle z_1', u_2\rangle)))$.

- RedProjFail Then $\mathsf{e}_1 = \pi_i(\varepsilon\,\langle \mathsf{v}_1,\mathsf{v}_2\rangle)$ and $\mathsf{e}_2 = \mathbf{error}$. Then combining TransformProj with Lemma 5.2 and Lemma 5.3, we have $\mathcal{E}[\![\pi_i\langle \mathsf{v}_1,\mathsf{v}_2\rangle]\!]k$ steps to $(\lambda x.\, \mathbf{let}\, y_1 := \pi_1 x \,\mathbf{in}\, \ldots)(\langle \mathcal{T}[\![\varepsilon]\!], \langle \mathcal{V}[\![\mathsf{v}_1]\!], \mathcal{V}[\![\mathsf{v}_2]\!]\rangle\rangle)$. If we β-reduce, and substitute for the let-expressions, we get $\mathrm{PROD}_i(\mathcal{T}[\![\varepsilon]\!], (\lambda z_1.\ldots))$. We then apply Lemma 5.1 and let-substitution to step to **error**.

- RedAscr Then $e_1 = \varepsilon_1\ (\varepsilon_2\ r)$ and $e_2 = (\varepsilon_1 \sqcap \varepsilon_2)\ r$. Applying TransformEv with Lemma 5.2 and and Lemma 5.3, we can see that this steps to $\mathrm{MEET}(\mathcal{T}[\![\varepsilon_1]\!], \mathcal{T}[\![\varepsilon_2]\!], (\lambda y.\ k(\langle y, u_2\rangle)))$ where $\mathcal{V}[\![\varepsilon_2\ r]\!] = \langle \mathcal{T}[\![\varepsilon_2]\!], u_2\rangle$. By Lemma 5.1, this steps to $k(\langle \mathcal{T}[\![\varepsilon_1 \sqcap \varepsilon_2]\!], u_2\rangle)$, which we can also step to from $\mathcal{E}[\![(\varepsilon_1 \sqcap \varepsilon_2)\ r]\!]k$ by Lemma 5.3.

- RedAscrFail Then $e_1 = \varepsilon_1\ (\varepsilon_2\ r)$ and $e_2 = \mathbf{error}$. Applying TransformEv with Lemma 5.2 and and Lemma 5.3, we can see that this steps to $\mathrm{MEET}(\mathcal{T}[\![\varepsilon_1]\!], \mathcal{T}[\![\varepsilon_2]\!], (\lambda y.\ k(\langle y, u_2\rangle)))$ where $\mathcal{V}[\![\varepsilon_2\ r]\!] = \langle \mathcal{T}[\![\varepsilon_2]\!], u_2\rangle$. By Lemma 5.1 along with our premise, this steps to **error**.

- RedContext: Then $e_1 = C[e_1']$ and $e_2 = C[e_2']$ where $e_1' \longrightarrow e_2'$. By our hypothesis, $\mathcal{E}[\![e_1]\!]k \equiv \mathcal{E}[\![e_2]\!]k$ for any k.

  Suppose that C is one of $\square\ e$, $(\square, e)$, $\pi_1\ \square$, $\pi_2\ \square$, $\varepsilon\ \square$, $\square + e$, $\square \stackrel{?}{=} e$ or **if** $\square$ **then** e_3 **else** e_4. These are the cases where the hole is the "first" sub-expression. In each case, there exists some k' such that $\mathcal{E}[\![C[e_1']]\!]k = \mathcal{E}[\![e_1']\!]k'$ and $\mathcal{E}[\![C[e_2']]\!]k = \mathcal{E}[\![e_2']\!]k'$. By our hypothesis, these terms are equal.

  The remaining cases are when the first sub-expression is already a value, and the context frame hole is the second sub-expression. In these cases, there exists some v (the first sub-expression) and k' such that $\mathcal{E}[\![C[e_1']]\!]k = \mathcal{E}[\![v]\!](\lambda x.\ \mathcal{E}[\![e_1']\!]k')$ and $\mathcal{E}[\![C[e_2']]\!]k = \mathcal{E}[\![v]\!](\lambda x.\ \mathcal{E}[\![e_2']\!]k')$. We assume the bound variable x is fresh, that is, it does not occur in e_1' or e_2'. We can apply Lemma 5.3 to show that these step to $[\mathcal{V}[\![v]\!]/x]\mathcal{E}[\![e_1']\!]k'$ and $[\mathcal{V}[\![v]\!]/x]\mathcal{E}[\![e_2']\!]k'$ respectively. Lemma 5.4 and our freshness assumption shows that these are equivalent to $\mathcal{E}[\![e_1']\!]([\mathcal{V}[\![v]\!]/x]k)$ and $\mathcal{E}[\![e_2']\!]([\mathcal{V}[\![v]\!]/x]k')$ respectively. Finally, our hypothesis shows that these two terms are equivalent.

- RedContextFail: then $e_1 = C[e_1']$ and $e_2 = \mathbf{error}$ where $e_1' \longrightarrow \mathbf{error}$. By our hypothesis, $\mathcal{E}[\![e_1']\!]k \equiv \mathbf{error}$ for any k.

  Suppose that C is one of $\square\ e$, $(\square, e)$, $\pi_1\ \square$, $\pi_2\ \square$, $\varepsilon\ \square$, $\square + e$, $\square \stackrel{?}{=} e$ or **if** $\square$ **then** e_3 **else** e_4. These are the cases where the hole is the "first" sub-expression. In each case, there exists some k' such that $\mathcal{E}[\![C[e_1']]\!]k = \mathcal{E}[\![e_1']\!]k'$. By our hypothesis, this steps to **error**.

  The remaining cases are when the first sub-expression is already a value, and the context frame hole is the second sub-expression. In these cases, there exists some v (the first sub-expression) and k' such that $\mathcal{E}[\![C[e_1']]\!]k = \mathcal{E}[\![v]\!](\lambda x.\ \mathcal{E}[\![e_1']\!]k')$. We assume the bound variable x is fresh, that is, it does not occur in e_1'. We can apply Lemma 5.3 to show that this steps to $[\mathcal{V}[\![v]\!]/x]\mathcal{E}[\![e_1']\!]k'$. Lemma 5.4 and our freshness assumption show that this is equivalent to $\mathcal{E}[\![e_1']\!]([\mathcal{V}[\![v]\!]/x]k')$, which, by our hypothesis, steps to **error**.

□

The conventional whole-program correctness theorem follows from this directly. If we take natural numbers as observables, we note that if $t \equiv n$, then $t \longrightarrow^* n$, since n cannot reduce further. Induction on the number of source-reduction steps gives us whole-program correctness.

Corollary 5.1 (Whole Program Correctness)**.** *If* $eval(\mathsf{e}) = n$*, then* $eval(\mathcal{E}[\![\mathsf{e}]\!](\lambda x.\ \mathbf{halt}\ [x])) = \mathbf{halt}\ [\mathcal{V}[\![n]\!]]$.

6 Breaking Full Abstraction

Unfortunately, our translation does not preserve contextual equivalence of source programs. Consider $(\lambda x.\ x{+}1)\ :\ \mathsf{Nat}{\rightarrow}\mathsf{Nat}$ and $(\lambda x.\ \{\mathsf{Nat}\}\ x{+}1)\ :\ \mathsf{Nat}{\rightarrow}\mathsf{Nat}$. The type rules of our language ensures that any value substituted for x must be a number, so annotating x with $\{\mathsf{Nat}\}$ has no effect. However, in our translation, such an annotation is translated into a meet operation between the annotation of x and NAT. In the context $\square(\langle \mathrm{TAG}, n\rangle)$, where TAG is not NAT or DYN, this meet operation will fail, cause the second expression to produce **error** while the first succeeds.

References

Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837670. URL `http://doi.acm.org/10.1145/2837614.2837670`.

Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *Proc. ACM Program. Lang.*, 3(POPL):17:1–17:30, January 2019. ISSN 2475-1421. doi: 10.1145/3290330. URL `http://doi.acm.org/10.1145/3290330`.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581484. URL `http://doi.acm.org/10.1145/581478.581484`.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999. ISSN 0164-0925. doi: 10.1145/319301.319345. URL `http://doi.acm.org/10.1145/319301.319345`.