FunTAL: Reasonably Mixing a Functional Language with Assembly [*]

Daniel Patterson
Northeastern University, USA
dbp@ccs.neu.edu

Jamie Perconti
Northeastern University, USA
jamieperconti@gmail.com

Christos Dimoulas
Harvard University, USA
chrdimo@seas.harvard.edu

Amal Ahmed
Northeastern University, USA
amal@ccs.neu.edu

Abstract

We present FunTAL, the first multi-language system to formalize safe interoperability between a high-level functional language and low-level assembly code while supporting compositional reasoning about the mix. A central challenge in developing such a multi-language is bridging the gap between assembly, which is staged into jumps to continuations, and high-level code, where subterms return a result. We present a *compositional* stack-based typed assembly language that supports *components*, comprised of one or more basic blocks, that may be embedded in high-level contexts. We also present a logical relation for FunTAL that supports reasoning about equivalence of high-level components and their assembly replacements, mixed-language programs with callbacks between languages, and assembly components comprised of different numbers of basic blocks.

CCS Concepts • **Theory of computation** → **Semantics and reasoning**; • **Software and its engineering** → **Formal language definitions**

Keywords multi-language semantics, typed assembly language, inline assembly, contextual equivalence, logical relations

[*] We use **blue** sans-serif to typeset our functional language $\mathbb{F}$ and **red** roman to typeset our typed assembly language $\mathbb{T}$. This paper will be much easier to follow if read/printed in color.

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062347

1. Introduction

Developers frequently integrate code written in lower-level languages into their high-level-language programs. For instance, OCaml and Haskell developers may leverage the FFI to make use of libraries implemented in C, while Rust developers may include inline assembly directly. In each of these cases, developers resort to the lower-level language so they can use features unavailable in the high-level language to gain access to hardware or fine-tune performance.

However, the benefits of mixed-language programs come at a price. To reason about the behavior of a high-level component, developers need to think not only about the semantics of the high-level language, but also about the way their high-level code was compiled and all interactions with low-level code. Since low-level code usually comes without safety guarantees, invalid instructions could crash the program. More insidiously, low-level code can potentially alter control flow, mutate values that should be inaccessible, or introduce security vulnerabilities that would not be possible in the high-level language. Unfortunately, there are no mixed-language systems that enable non-expert programmers to reason about interactions with lower-level code—i.e., systems that guarantee safe interoperability and provide rules for compositional reasoning in a mixed-language setting.

Even if developers don't directly write inline assembly, mixed-language programs are a reality that compiler writers and compiler-verification efforts must contend with. For instance, mixed programs show up in modern just-in-time (JIT) compilers, where the high-level language is initially interpreted until the runtime can identify portions to statically compile, at which point those portions of the code are replaced with equivalent assembly. These assembly components will include hooks to move back into the interpreted runtime, corresponding closely to the semantics of a mixed-language program. Verifying correctness of such JITs requires proving that the high-level fragment and its compiled replacement are *contextually equivalent* in the mixed language. Contextual equivalence guarantees that in any whole program replacing the high-level fragment with the compiled version will not change the behavior of the program.

In the case of traditional compilers, compiled components are frequently linked with target code compiled from a different source language, or with low-level routines that form part of the runtime system. Perconti and Ahmed [1, 22] argue that correctness theorems for verified compilers that account for such linking must include mixed-language reasoning. Specifically, they set up multi-languages that specify the rules of source-target interoperability and then express compiler correctness as multi-language equivalence between a source component e_S and its compiled version e_T. Hence, the theorem ensures that e_T linked with some arbitrary target code e'_T will behave the same as e_S interoperating with e'_T.

All of the above scenarios call for the design of a multi-language that specifies interoperability between a high-level language and assembly, along with proof methods for reasoning about equivalence of components in this setting. Note that Perconti and Ahmed [22] left the design of a multi-language that embeds assembly as future work. Since they did not show how to verify a code generation pass to assembly, they didn't need to define interoperability between a high-level, expression-based language and a low-level language with direct jumps.

In this paper, we present FunTAL, a multi-language system that allows assembly to be embedded in a typed functional language and vice versa. A key difficulty is ensuring that the embedded assembly has local and well-controlled effects. This is challenging because assembly is inherently *non-compositional*—control can change to an arbitrary point with direct jumps and code can access arbitrary values far up on the call-stack. To allow a compositional functional language to safely interoperate with assembly, such behavior must be constrained, which we do using types at the assembly level. Moreover, we need to identify the right notion of *component* in assembly: intuitively, an assembly component may be comprised of multiple basic blocks and we should be able to show equivalence between terms of the functional language (i.e., high-level components) and multi-block assembly components. But how do we identify which blocks should be grouped together into a component without imposing so much high-level structure on assembly that it ceases to be low level? Even once we identify such groupings, we must still contend with the control-flow gap between a direct-style functional language in which terms return results and assembly code that is staged into jumps to continuations. Finally, we must find a way to embed functional code in assembly so we can support callbacks from assembly to the functional language.

Contributions We make the following contributions:

- We design a compositional typed assembly language (TAL) called $\mathbb{T}$, building on the stack-based typed assembly language of Morrisett *et al.* [18] (henceforth, STAL). The central novelty of our TAL $\mathbb{T}$ are extensions to an STAL-like type system that help us reason about multi-block components and bridge the gap between direct-style high-level components and continuation-based assembly components (§3).
- We present a multi-language $\mathbb{FT}$ in the style of Matthews-Findler [16] that supports interoperability between a simply typed functional language $\mathbb{F}$ with recursive types and our TAL $\mathbb{T}$ (§4).
- We develop a novel step-indexed Kripke logical relation for reasoning about equivalence of $\mathbb{FT}$ components (§5). It builds on prior logical relations for mutable state [4, 10, 22], but is the first to support reasoning about equivalence of programs that mix assembly with lambdas (including callbacks between them), and of assembly components comprised of different numbers of basic blocks. The central novelty lies in the mechanics of accommodating assembly and equivalence of multi-block components.

The technical appendix [21] includes complete language semantics, definitions, and proofs, some of which are elided in this paper. Our artifact provides an in-browser type checker and machine stepper for the multi-language to aid understanding and experimentation with $\mathbb{FT}$ programs. The artifact, available at `https://dbp.io/artifacts/funtal`, includes runnable versions of all examples in the paper.

2. Main Ingredients of the Mix

We design a *compositional* TAL $\mathbb{T}$ that draws largely from Morrisett *et al.*'s STAL [18], which has a single explicit stack and assembly instructions to allocate, read, write, and free stack cells. We follow much of their basic design, including the use of stack-tail polymorphism to hide values on the stack so they will be preserved across calls, and the use of register-file and stack typing to specify preconditions for jumping to a code block.

Our main novelty is identifying the notion of a TAL *component*. In $\mathbb{T}$, we need to be able to reason about a component $\mathbf{e_T}$ because we will eventually be embedding these components as terms in a high-level functional language called $\mathbb{F}$. A component $\mathbf{e_T}$ must be composed of assembly instructions, but we don't want to restrict it to a single basic block so we use a pair $(\mathbf{I}, \mathbf{H})$ of an instruction sequence $\mathbf{I}$ and a local heap fragment $\mathbf{H}$ that maps locations to code blocks used in local intra-component jumps.

The combined language $\mathbb{FT}$ is a typical Matthews-Findler multi-language [16], where the syntax of both languages are combined and boundary terms are added to mediate interactions between the two. A boundary term ${}^{\tau}\mathcal{FT}\,\mathbf{e_T}$ means that the $\mathbb{T}$ component $\mathbf{e_T}$ within the boundary will be used in an $\mathbb{F}$ context at type $\boldsymbol{\tau}$. To be well-typed, the inner component $\mathbf{e_T}$ should have the type translated from $\boldsymbol{\tau}$ according to the multi-language type translation in §4.

$\mathbb{FT}$ exists to enable reasoning about the equivalence of $\mathbb{F}$ expressions and $\mathbb{T}$ components, or mixed combinations of the two. Intuitively, we would like to treat blocks of assembly as similar to functions in high-level languages. Semantically, functions are objects that, given related inputs, produce related outputs. Following STAL we can, at least, model the state of the stack and a subset of the registers as inputs. But blocks of assembly instructions do not have clear outputs to relate, leading us towards one of our central novel contributions.

In STAL, every basic block has type $\forall[\Delta].\{\chi;\sigma\}$, where Δ contains type parameters, and χ and σ are respectively the register and stack typing preconditions. Since every block is in continuation style, blocks never return, always jumping to the next block, so there never need be outputs to relate — the output of a block is just the input constraints on the block to which it jumps. In our mixed-language setting we must, therefore, provide components with return continuations which when called from high-level code contain a halting instruction, and when called from assembly jump to the next step in execution. In order to determine the result type—i.e., the type of the value that is either halted with or passed to the next block—we extend the STAL code pointer type to $\forall[\Delta].\{\chi;\sigma\}^{\mathrm{q}}$, where q is our critical addition.

A *return marker* q specifies the register or stack position where the return continuation is stored. This allows us, following a basic calling convention, to determine the type of the value that will be passed to that continuation. As we will see in later sections, there are a few other forms that q can take, but they all support our ability to reason about $\mathbb{T}$ components as semantic objects that produce values of a specific type. This allows us to reason not only about the equivalence of structurally different assembly components made up of different numbers of basic blocks, but of components made up of entirely different mixes of languages.

3. Typed Assembly Language: $\mathbb{T}$

Syntax Figure 1 presents the full syntax of $\mathbb{T}$, our typed assembly language. Value types τ are the types ascribed to values small enough to fit in a register, including base values, recursive and existential types, and mutable ($\mathbf{ref}$) or immutable ($\mathbf{box}$) pointers to heap values. We ascribe value types τ to word values w, which include unit $()$, integers n, locations ℓ, existential $\mathbf{pack}$s, and recursive $\mathbf{fold}$s. We additionally follow STAL's convention that a word value w applied to a type instantiation ω is itself a value $\mathrm{w}[\omega]$. Small values u include word values w, but also can be a register r that contains a word value. Instructions accept small values u as operands; hence, in the operational semantics, if u is a register we first load the value from the register, while if u is a word value we use it directly.

We ascribe heap-value types ψ to heap values h. These include tuples of word values $\langle \mathrm{w},\ldots,\mathrm{w}\rangle$ and code blocks $\mathbf{code}[\Delta]\{\chi;\sigma\}^{\mathrm{q}}.\mathrm{I}$, which have types $\langle\tau,\ldots,\tau\rangle$ and

Value type τ	::=	$\alpha \mid \mathbf{unit} \mid \mathbf{int} \mid \exists\alpha.\tau \mid \mu\alpha.\tau$ $\mid \mathbf{ref}\ \langle\tau,\ldots,\tau\rangle \mid \mathbf{box}\ \psi$
Word value w	::=	$() \mid \mathrm{n} \mid \ell \mid \mathbf{pack}\langle\tau,\mathrm{w}\rangle\ \mathbf{as}\ \exists\alpha.\tau$ $\mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathrm{w} \mid \mathrm{w}[\omega]$
Register r	::=	$\mathtt{r1} \mid \mathtt{r2} \mid \cdots \mid \mathtt{r7} \mid \mathtt{ra}$
Small value u	::=	$\mathrm{w} \mid \mathrm{r} \mid \mathbf{pack}\langle\tau,\mathrm{u}\rangle\ \mathbf{as}\ \exists\alpha.\tau$ $\mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathrm{u} \mid \mathrm{u}[\omega]$
Type instantiation ω	::=	$\tau \mid \sigma \mid \mathrm{q}$
Heap value type ψ	::=	$\forall[\Delta].\{\chi;\sigma\}^{\mathrm{q}} \mid \langle\tau,\ldots,\tau\rangle$
Heap value h	::=	$\mathbf{code}[\Delta]\{\chi;\sigma\}^{\mathrm{q}}.\mathrm{I} \mid \langle\mathrm{w},\ldots,\mathrm{w}\rangle$
Register typing χ	::=	$\cdot \mid \chi, \mathrm{r} : \tau$
Stack typing σ	::=	$\zeta \mid \bullet \mid \tau :: \sigma$
Return marker q	::=	$\mathrm{r} \mid \mathrm{i} \mid \epsilon \mid \mathbf{end}\{\tau;\sigma\}$
Type env Δ	::=	$\cdot \mid \Delta,\alpha \mid \Delta,\zeta \mid \Delta,\epsilon$
Heap typing Ψ	::=	$\cdot \mid \Psi, \ell :^{\nu} \psi$ where ν ::= $\mathbf{ref} \mid \mathbf{box}$
Memory M	::=	$(\mathrm{H},\mathrm{R},\mathrm{S})$
Heap fragment H	::=	$\cdot \mid \mathrm{H}, \ell \mapsto \mathrm{h}$
Register file R	::=	$\cdot \mid \mathrm{R}, \mathrm{r} \mapsto \mathrm{w}$
Stack S	::=	$\mathbf{nil} \mid \mathrm{w} :: \mathrm{S}$
Instruction sequence I	::=	
$\iota;\mathrm{I}$		instruction sequencing
$\mathtt{jmp}\ \mathrm{u}$		jump to u within same component
$\mathtt{call}\ \mathrm{u}\ \{\sigma,\mathrm{q}\}$		jump to u, with return address at q
$\mathtt{ret}\ \mathrm{r}\ \{\mathrm{r_r}\}$		jump back to code at r with result in $\mathrm{r_r}$
$\mathtt{halt}\ \tau,\sigma\ \{\mathrm{r_r}\}$		halt with value type τ in register $\mathrm{r_r}$
Single instruction ι	::=	
$\mathtt{aop}\ \mathrm{r_d},\mathrm{r_s},\mathrm{u}$		store result of $\mathtt{add}\vert\mathtt{mul}\vert\mathtt{sub}$ in $\mathrm{r_d}$
$\mathtt{bnz}\ \mathrm{r},\mathrm{u}$		jump to u if r contains 0
$\mathtt{ld}\ \mathrm{r_d},\mathrm{r_s}[\mathrm{i}]$		load from ith position in tuple at $\mathrm{r_s}$
$\mathtt{st}\ \mathrm{r_d}[\mathrm{i}],\mathrm{r_s}$		store to ith position in mutable tuple at $\mathrm{r_d}$
$\mathtt{ralloc}\ \mathrm{r_d},\mathrm{n}$		alloc mutable n-tuple from stack
$\mathtt{balloc}\ \mathrm{r_d},\mathrm{n}$		alloc immutable n-tuple from stack
$\mathtt{mv}\ \mathrm{r_d},\mathrm{u}$		move value u into register $\mathrm{r_d}$
$\mathtt{salloc}\ \mathrm{n}$		allocate n stack cells with unit values
$\mathtt{sfree}\ \mathrm{n}$		free n stack cells
$\mathtt{sld}\ \mathrm{r_d},\mathrm{i}$		load ith stack value into $\mathrm{r_d}$
$\mathtt{sst}\ \mathrm{i},\mathrm{r_s}$		store $\mathrm{r_s}$ into ith stack slot
$\mathtt{unpack}\ \langle\alpha,\mathrm{r_d}\rangle\ \mathrm{u}$		unpack existential, binding to $\alpha,\mathrm{r_d}$
$\mathtt{unfold}\ \mathrm{r_d},\mathrm{u}$		unfold recursive type
Component e	::=	(I,H)
Halt instruction v	::=	$\mathtt{halt}\ \tau,\sigma\ \{\mathrm{r_r}\}$
Evaluation context E	::=	$([\cdot],\cdot)$

Figure 1. $\mathbb{T}$ Syntax

$\forall[\Delta].\{\chi;\sigma\}^{q}$, respectively. Note that we have mutable ($\mathbf{ref}$) references to tuples but only immutable ($\mathbf{box}$) references to code, since we prohibit self-modifying code.

Code blocks $\mathbf{code}[\Delta]\{\chi;\sigma\}^{q}.\mathbf{I}$ specify a type environment Δ, a register file typing χ, and a stack type σ for an instruction sequence $\mathbf{I}$. Here χ and σ are preconditions for safely jumping to $\mathbf{I}$: χ is a mapping from registers $\mathbf{r}$ to the type of values τ the registers must contain, while σ is a list of value types on top of the stack that may end with an abstract stack-tail variable ζ. The type variables in Δ, which may appear free in χ, σ, and $\mathbf{I}$, must be instantiated when we jump to the code block. If this code block is stored at location ℓ, and register $\mathbf{r}$ contains ℓ, we can jump to it via $\mathtt{jmp}\;\mathbf{r}[\overline{\omega}]$ where $\overline{\omega}$ instantiates the variables in Δ. (We use vector notation, e.g., $\overline{\omega}$ or $\overline{\tau}$, to denote a sequence.)

As discussed in §2, our code blocks include a novel return marker q, which tells us where to find the current return continuation. Here q can be a register $\mathbf{r}$ in χ, or a stack index $\mathbf{i}$ that is accessible in σ (i.e., the $\mathbf{i}$th stack slot is not hidden in the stack tail ζ). Return markers can also range over type variables ϵ which we use to abstract over return markers (as we explain below). There is also a special return marker $\mathbf{end}\{\tau;\sigma\}$ which means that when the current component finishes it should halt with a value of type τ and stack of type σ. In $\mathbb{T}$, this would mean the end of the program with a $\mathtt{halt}$ instruction, but within a multi-language boundary the same $\mathtt{halt}$ results in a transition to the high-level language.

A memory $\mathbf{M}$ includes a heap $\mathbf{H}$ which maps locations ℓ to heap values $\mathbf{h}$, a register file $\mathbf{R}$ which maps registers $\mathbf{r}$ to word values $\mathbf{w}$, and a stack $\mathbf{S}$ which is a list of word values.

An instruction sequence $\mathbf{I}$ is a list of instructions terminated by one of three jump instructions ($\mathtt{jmp}$, $\mathtt{call}$, $\mathtt{ret}$) or the $\mathtt{halt}$ instruction. The distinction between jump instructions is a critical part of $\mathbb{T}$ explored in depth later in this section. Individual instructions ι include many standard assembly instructions and are largely similar to STAL.

A component $\mathbf{e}$ is a tuple $(\mathbf{I}, \mathbf{H})$ of instructions $\mathbf{I}$ and a local heap fragment $\mathbf{H}$. The local heap fragment can contain multiple local blocks used by the component. We distinguish the $\mathtt{halt}$ instruction as a value $\mathbf{v}$, as it is the only $\mathbb{T}$ instruction sequence that does not reduce.

Operational Semantics We specify a small-step operational semantics as a relation on memories $\mathbf{M}$ and components $\mathbf{e}$: $\langle \mathbf{M} \mid \mathbf{e}\rangle \longmapsto \langle \mathbf{M}' \mid \mathbf{e}'\rangle$. Operationally, we merge local heap fragments to the global heap and then use the evaluation context $\mathbf{E}$ to reduce instructions according to relation: $\langle \mathbf{M} \mid \mathbf{I}\rangle \longmapsto \langle \mathbf{M}' \mid \mathbf{I}'\rangle$. In $\mathbb{T}$, evaluation contexts $\mathbf{E}$ are not particularly interesting, but when $\mathbb{T}$ is embedded within the multi-language, $\mathbf{E}$ will include boundaries. While Figure 1 includes operational descriptions of the instructions, the full semantics are standard and elided.

Type System In Figure 2 we present a selection of typing rules for $\mathbb{T}$. We elide various type judgments and well-formedness judgments for small values, heap fragments, and

$$\boxed{\Psi;\Delta;\chi;\sigma;q \vdash \iota \Rightarrow \Delta';\chi';\sigma';q'} \quad \text{where } \cdot[\Delta];\chi;\sigma \vdash q$$

$$\frac{\Psi;\Delta;\chi \vdash u : \tau \qquad q \neq r_d \qquad u \neq q}{\Psi;\Delta;\chi;\sigma;q \vdash \mathtt{mv}\; r_d, u \Rightarrow \Delta;\chi[r_d : \tau];\sigma;q}$$

$$\frac{\chi(r_s) = \tau}{\Psi;\Delta;\chi;\sigma;r_s \vdash \mathtt{mv}\; r_d, r_s \Rightarrow \Delta;\chi[r_d : \tau];\sigma;r_d}$$

$$\boxed{\Psi;\Delta;\chi;\sigma;q \vdash I} \quad \text{where } \cdot[\Delta];\chi;\sigma \vdash q$$

$$\frac{\Psi;\Delta;\chi;\sigma;q \vdash \iota \Rightarrow \Delta';\chi';\sigma';q' \qquad \Psi;\Delta';\chi';\sigma';q' \vdash I}{\Psi;\Delta;\chi;\sigma;q \vdash \iota;I}$$

$$\frac{\chi(r) = \tau}{\Psi;\Delta;\chi;\sigma;\mathbf{end}\{\tau;\sigma\} \vdash \mathtt{halt}\;\tau,\sigma\;\{r\}}$$

$$\frac{\Psi;\Delta;\chi \vdash u : \mathbf{box}\,\forall[].\{\chi';\sigma\}^{q} \qquad \Delta \vdash \chi \leq \chi'}{\Psi;\Delta;\chi;\sigma;q \vdash \mathtt{jmp}\;u}$$

$$\frac{\chi(r) = \mathbf{box}\,\forall[].\{r' : \tau;\sigma\}^{q'} \qquad \chi(r') = \tau}{\Psi;\Delta;\chi;\sigma;r \vdash \mathtt{ret}\;r\;\{r'\}}$$

$$\frac{\begin{array}{c} \Psi;\Delta;\chi \vdash u : \mathbf{box}\,\forall[\zeta,\epsilon].\{\hat{\chi};\hat{\sigma}\}^{\hat{q}} \qquad \Delta \vdash \hat{\chi} \setminus \hat{q} \\ \text{ret-addr-type}(\hat{q},\hat{\chi},\hat{\sigma}) = \mathbf{box}\,\forall[].\{r : \tau;\hat{\sigma}'\}^{\epsilon} \\ \Delta \vdash \tau \qquad \Delta \vdash \hat{\sigma}'[\sigma_0/\zeta] \\ \Delta \vdash \forall[].\{\hat{\chi}[\sigma_0/\zeta][\mathbf{end}\{\tau^*;\sigma^*\}/\epsilon];\hat{\sigma}[\sigma_0/\zeta][\mathbf{end}\{\tau^*;\sigma^*\}/\epsilon]\}^{\hat{q}} \\ \Delta \vdash \chi \leq \hat{\chi}[\sigma_0/\zeta][\mathbf{end}\{\tau^*;\sigma^*\}/\epsilon] \\ \sigma = \overline{\tau} :: \sigma_0 \qquad \hat{\sigma} = \overline{\tau} :: \zeta \qquad \hat{\sigma}' = \overline{\tau'} :: \zeta \end{array}}{\Psi;\Delta;\chi;\sigma;\mathbf{end}\{\tau^*;\sigma^*\} \vdash \mathtt{call}\;u\;\{\sigma_0,\mathbf{end}\{\tau^*;\sigma^*\}\}}$$

$$\frac{\begin{array}{c} \Psi;\Delta;\chi \vdash u : \mathbf{box}\,\forall[\zeta,\epsilon].\{\hat{\chi};\hat{\sigma}\}^{\hat{q}} \\ \Delta \vdash \hat{\chi} \setminus \hat{q} \qquad \text{ret-addr-type}(\hat{q},\hat{\chi},\hat{\sigma}) = \forall[].\{r : \tau;\hat{\sigma}'\}^{\epsilon} \\ \Delta \vdash \tau \qquad \Delta \vdash \hat{\sigma}'[\sigma_0/\zeta] \\ \Delta \vdash \forall[].\{\hat{\chi}[\sigma_0/\zeta][i{+}k{-}j/\epsilon];\hat{\sigma}[\sigma_0/\zeta][i{+}k{-}j/\epsilon]\}^{\hat{q}} \\ \Delta \vdash \chi \leq \hat{\chi}[\sigma_0/\zeta][i{+}k{-}j/\epsilon] \\ \sigma = \tau_0 :: \cdots :: \tau_j :: \sigma_0 \qquad \hat{\sigma} = \tau_0 :: \cdots :: \tau_j :: \zeta \\ j < i \qquad \hat{\sigma}' = \tau_0' :: \cdots :: \tau_k' :: \zeta \end{array}}{\Psi;\Delta;\chi;\sigma;i \vdash \mathtt{call}\;u\;\{\sigma_0, i{+}k{-}j\}}$$

$$\boxed{\Psi;\Delta;\chi;\sigma;q \vdash e : \tau;\sigma'}$$

$$\frac{\begin{array}{c} \Psi \vdash H : \Psi' \qquad \forall(\ell :^{\nu} \psi) \in \Psi.\ \nu = \mathbf{box} \\ \text{ret-type}(q,\chi,\sigma) = \tau;\sigma' \qquad (\Psi,\Psi');\Delta;\chi;\sigma;q \vdash I \end{array}}{\Psi;\Delta;\chi;\sigma;q \vdash (I,H) : \tau;\sigma'}$$

$$\begin{array}{l} \text{ret-type}(r,\chi,\sigma) = \tau;\sigma' \text{if } \chi(r) = \mathbf{box}\,\forall[].\{r' : \tau;\sigma'\}^{q} \\ \text{ret-type}(i,\chi,\sigma) = \tau;\sigma' \text{if } \sigma(i) = \mathbf{box}\,\forall[].\{r' : \tau;\sigma'\}^{q} \\ \text{ret-type}(\mathbf{end}\{\tau;\sigma'\},\chi,\sigma) = \tau;\sigma' \\ \text{ret-addr-type}(r,\chi,\sigma) = \forall[].\{r' : \tau;\sigma'\}^{q'} \\ \qquad \text{if } \chi(r) = \mathbf{box}\,\forall[].\{r' : \tau;\sigma'\}^{q'} \\ \text{ret-addr-type}(i,\chi,\sigma) = \forall[].\{r' : \tau;\sigma'\}^{q'} \\ \qquad \text{if } \sigma(i) = \mathbf{box}\,\forall[].\{r' : \tau;\sigma'\}^{q'} \end{array}$$

Figure 2. Selected $\mathbb{T}$ Typing Rules

register files as they are standard, focusing instead on novel rules for instructions, instruction sequences, and components. Full details appear in our technical appendix [21].

Instructions ι and instruction sequences $\mathbf{I}$ are typed under a static heap Ψ, a type environment Δ, a register file typing χ, a stack typing σ, and return marker $\mathtt{q}$. An instruction ι may change any of these except the static heap. Critically, the instruction and instruction-sequence judgments impose restrictions on the return marker $\mathtt{q}$ (written $\cdot[\Delta];\chi;\sigma\vdash\mathtt{q}$) to ensure that a block of instructions knows to where it is returning. This means that $\mathtt{q}$ cannot be ϵ and if $\mathtt{q}$ is a register or stack index its type should be visible in χ or σ. The judgment $\Delta'[\Delta];\chi;\sigma\vdash\mathtt{q}$ ensures that if $\mathtt{q}$ is ϵ, then ϵ is in Δ not Δ', which in this case is empty.[1] It also checks that we can look up the types expected by the return continuation at $\mathtt{q}$ using $\text{ret-type}(\mathtt{q},\chi,\sigma)$ (see bottom of Figure 2).

The $\mathtt{mv}$ instruction shown in Figure 2 has two cases. In the first case, we are loading a small value $\mathbf{u}$ with type τ into register $\mathtt{r_d}$, which we know is not the return marker $\mathtt{q}$. We also restrict $\mathbf{u}$ to not be the current return marker $\mathtt{q}$, as in that case the second $\mathtt{mv}$ rule described below must be used. After the $\mathtt{mv}$, the register-file typing now reflects the updated register, which we write as $\chi[\mathtt{rd}:\tau]$, and no other changes have occurred. The second case is that we are moving the value in register $\mathtt{r_s}$ into register $\mathtt{r_d}$, where $\mathtt{r_s}$ is the current return marker so it is pointing to the return continuation. In this case, not only do we update the register file, we also change the return marker to reflect that the continuation is now in $\mathtt{r_d}$. Other instructions, like $\mathtt{sst}$ and $\mathtt{sld}$, similarly have cases depending on whether the operation will change where the return continuation is stored.

Instruction typing judgments are lifted to instruction sequences by matching the postcondition of the instruction at the head of the list to the precondition of the rest of the sequence, as shown in Figure 2. We illustrate how sequences are type-checked with the following small example. Note that each instruction's postcondition is used as the precondition of the next.

$$\begin{aligned}
\cdot;\cdot;\cdot;\bullet;\mathtt{ra}\vdash \mathtt{mv}\,\mathtt{r1},\mathbf{42};&\Rightarrow\ \cdot;\mathtt{r1}:\mathbf{int};\bullet;\mathtt{ra}\\
\mathtt{salloc}\,\mathbf{1};&\Rightarrow\ \cdot;\mathtt{r1}:\mathbf{int};\mathbf{unit}::\bullet;\mathtt{ra}\\
\mathtt{sst}\,\mathbf{0},\mathtt{r1};&\Rightarrow\ \cdot;\mathtt{r1}:\mathbf{int};\mathbf{int}::\bullet;\mathtt{ra}
\end{aligned}$$

First, we load $\mathbf{42}$ into register $\mathtt{r1}$, which is reflected in the register file typing $\mathtt{r1}:\mathbf{int}$. We then allocate one cell on the stack, which starts out as $\mathbf{unit}$. Now that there is space, we can store the value of register $\mathtt{r1}$ into the $\mathbf{0}$th slot on the stack, which is then reflected in the stack typing.

Next in Figure 2, we show the $\mathtt{halt}$ instruction, which requires the $\mathbf{end}\{\tau;\sigma\}$ return marker, indicating the type of the value in the register specified and the type of the stack. This instruction is how $\mathbb{T}$ programs terminate; in our $\mathbb{FT}$ multi-language, this will also be how a $\mathbb{T}$ component transfers a value back to a wrapping $\mathbb{F}$ component.

Next are the three jump instructions. First is the *intra-component jump* $\mathtt{jmp}$ instruction. This requires that the location $\mathbf{u}$ being jumped to be a code pointer (of type $\mathbf{box}\,\forall[].\{\chi';\sigma\}^{\mathtt{q}}$) that has preconditions χ' and σ for the register file and stack respectively, and return marker $\mathtt{q}$. The current register file χ must be a subset of the expected χ', which means that we can have more registers with values in them, but the types of registers that occur in χ' must match.

We also, critically, require that the return marker $\mathtt{q}$ on the code block being jumped to be the same as the current return marker. This captures the intuition of an intra-component jump. As noted before, blocks being jumped to must have fully instantiated return markers—informally, blocks cannot abstract over their own return markers. This restriction is only on instruction sequences; a component can have local blocks with abstract return markers. Consider the code pointer type:

$$\mathbf{box}\,\forall[\epsilon].\{\mathtt{ra}:\mathbf{box}\,\forall[].\{\mathtt{r1}:\tau;\sigma\}^{\epsilon};\sigma\}^{\mathtt{ra}}$$

This type is a pointer to a code block with a return marker type parameter ϵ that requires a stack of type σ and for register $\mathtt{ra}$, the return marker, to be a code pointer. This inner code pointer is the continuation, as the entire block has $\mathtt{ra}$ as its return marker, but the return marker for this continuation is ϵ. When the continuation in $\mathtt{ra}$ is jumped to it requires that the stack still have type σ and that a value of type τ be stored in register $\mathtt{r1}$. Since code pointers can't be jumped to until all their type variables are instantiated, the caller of this whole code block must provide a concrete continuation in register $\mathtt{ra}$ and instantiate ϵ with the corresponding concrete return marker before jumping.

As a concrete example consider the following well-typed $\mathtt{jmp}$ instruction:

$$\begin{aligned}
&\ell:{}^{\mathbf{box}}\forall[].\{\mathtt{r2}:\mathbf{unit};\mathbf{int}::\bullet\}^{\mathbf{end}\{\mathbf{unit};\bullet\}};\cdot;\\
&\quad\mathtt{r1}:\mathbf{int},\mathtt{r2}:\mathbf{unit};\mathbf{int}::\bullet;\mathbf{end}\{\mathbf{unit};\bullet\}\vdash\mathtt{jmp}\,\ell
\end{aligned}$$

As required, the $\mathtt{jmp}$ is to a code block ℓ that has the same return marker $\mathbf{end}\{\mathbf{unit};\bullet\}$. The current registers has $\mathtt{r1}$ set, which the block does not require, but also has the register $\mathtt{r2}$ set that the block does require. Finally, the stack type $\mathbf{int}::\bullet$ matches what the block expects. Note that since the stack currently has an $\mathbf{int}$ on it but the return marker says the stack must be empty, we will have to pop the integer off the stack either in the block ℓ or in some subsequent block that we jump to from ℓ before we $\mathtt{halt}$.

The next instruction in Figure 2 is $\mathtt{ret}$, which is the *inter-component* jump for returning from a component. Notably, the location being jumped to must be in a register; if it were still on the stack the type of σ would include itself. We require, first, that the register $\mathtt{r}$ being jumped to points to a code block with no type variables, and second that the register $\mathtt{r}'$ map to type τ, as required by the block being returned to. This is a type-enforced calling convention for

[1] Code pointers can have ϵ in the return marker of their return continuation, but by the time they are jumped to this ϵ must be instantiated. An example of this is shown later in this section.

the return value. Importantly, we make no restriction on the return marker $\mathbf{q}'$ on the block being jumped to. This is because with `ret` we are jumping back to a different component, which will in turn have its own return marker.

The last two typing rules shown in Figure 2 are for the `call` instruction which is our other *inter-component* jump. The first applies when the current component will terminate by `halt`ing. The second applies when the current component will terminate by jumping to another $\mathbb{T}$ component.

In some assembly languages, there is a convention that certain registers ("callee-saved") will be preserved such that when a `call` returns, those registers have the same values as before. However, we follow STAL in protecting values solely through stack-tail polymorphism, where a value can be stored in a part of the stack that has been abstracted away as a type variable. Static typing ensures that a callee that tried to read, write, or free values within the abstract tail would not type check. Values that are accessible can be passed in front of the abstract tail, and the callee is free to allocate values in front, but typing constraints may force them to free the values before returning.

As a concrete example of the first typing rule, consider the following well-typed `call` instruction:

$$\begin{aligned}&\ell\colon{}^{\mathbf{box}}\forall[\zeta,\epsilon].\{\mathtt{ra}\colon \mathbf{box}\,\forall[].\{\mathtt{r1}\colon\mathbf{int};\zeta\}^{\epsilon};\mathbf{unit}::\zeta\}^{\mathtt{ra}}\ ;\ \cdot\ ;\\ &\mathtt{r1}\colon\mathbf{int},\mathtt{ra}\colon\mathbf{box}\,\forall[].\{\mathtt{r1}\colon\mathbf{int};\mathbf{int}::\bullet\}^{\mathbf{end}\{\mathbf{int};\bullet\}}\ ;\\ &\mathbf{unit}::\mathbf{int}::\bullet\ ;\ \mathbf{end}\{\mathbf{unit};\bullet\}\\ &\qquad\vdash \mathtt{call}\,\ell\,\{\mathbf{int}::\bullet,\mathbf{end}\{\mathbf{int};\bullet\}\}\end{aligned}$$

We focus here on the stack and return continuation. The `call` instruction specifies a tail $\mathbf{int}::\bullet$ to protect. The block at ℓ being jumped to must have a stack that has the same front and an abstract tail, here $\mathbf{unit}::\zeta$. Further, the block being jumped to must return to a continuation (here stored at `ra`) with an abstract return marker ϵ. Once ϵ is instantiated with $\mathbf{end}\{\mathbf{int};\bullet\}$ the return continuation must match the current register file typing.

In the second `call` typing rule, the return marker is a stack position $\mathbf{i}$. The index $\mathbf{i}$ must be greater than the number of entries $\mathbf{j}$ on the input stack σ in front of the tail σ_0 specified in the instruction. The location being jumped to, $\mathbf{u}$, must be a code pointer with input registers $\hat{\chi}$ and stack $\hat{\sigma}$. Note that the prefix of $\hat{\sigma}$ matches the prefix of σ, $\tau_0::\cdots::\tau_{\mathbf{j}}$, but $\hat{\sigma}$ has the abstract tail ζ.

The final formal parameter to `call`, $\mathbf{i}+\mathbf{k}-\mathbf{j}$, is the return marker that the continuation for $\mathbf{u}$ must use. In particular, this is computed by taking the starting stack position $\mathbf{i}$ and then noting how the stack is modified between the input stack $\hat{\sigma}$ and output stack $\hat{\sigma}'$ by the code block pointed to by $\mathbf{u}$. After the call, the stack has $\mathbf{k}$ values in front but we know that position $\mathbf{i}$ was beyond the exposed $\mathbf{j}$ values, so the value on the stack at position $\mathbf{i}$ is now at position $\mathbf{i}+\mathbf{k}-\mathbf{j}$.

The fact that ret-addr-type$(\hat{\mathbf{q}},\hat{\chi},\hat{\sigma})$ is $\forall[].\{\mathbf{r}\colon\tau;\hat{\sigma}'\}^{\epsilon}$ ensures that the block being jumped to has a return continuation where a value of type τ is stored in some register, the stack has type $\hat{\sigma}'$, and the return marker is ϵ. Operationally $\mathbf{u}$ will get instantiated with $\mathbf{i}+\mathbf{k}-\mathbf{j}$ for ϵ, which based on the form of $\hat{\sigma}'$ means that the return continuation has preserved the original return location.

The register file subtyping constraint

$$\Delta\vdash\chi\leq\hat{\chi}[\sigma_0/\zeta][\mathbf{i}{+}\mathbf{k}{-}\mathbf{j}/\epsilon]$$

ensures that the current register type χ is a subtype of the target $\hat{\chi}$ once it has been concretely instantiated with the stack tail and return address.

We similarly check with

$$\Delta\vdash\forall[].\{\hat{\chi}[\sigma_0/\zeta][\mathbf{i}{+}\mathbf{k}{-}\mathbf{j}/\epsilon];\hat{\sigma}[\sigma_0/\zeta][\mathbf{i}{+}\mathbf{k}{-}\mathbf{j}/\epsilon]\}^{\hat{\mathbf{q}}}$$

that the code block type is well-formed when concretely instantiated, and with $\Delta\vdash\hat{\sigma}'[\sigma_0/\zeta]$ that the resulting stack is well-formed once concretely instantiated. Finally, we ensure with $\Delta\vdash\hat{\chi}\setminus\hat{\mathbf{q}}$ that if $\hat{\mathbf{q}}$ is a register then $\hat{\chi}$ is well-formed without it. This means that while $\hat{\mathbf{q}}$ may have free type variables ϵ and ζ, the rest of $\hat{\chi}$ cannot.

$$\begin{aligned}&\mathbf{f}=(\mathtt{mv}\,\mathtt{ra},\ell_{\mathbf{1ret}};\mathtt{call}\,\ell_1\,\{\bullet,\mathbf{end}\{\mathbf{int};\bullet\}\},\mathbf{H})\\ &\mathbf{H}(\ell_1)=\mathbf{code}[\zeta,\epsilon]\{\mathtt{ra}\colon\forall[].\{\mathtt{r1}\colon\mathbf{int};\zeta\}^{\epsilon};\zeta\}^{\mathtt{ra}}.\\ &\qquad\mathtt{salloc}\,1;\mathtt{sst}\,0,\mathtt{ra};\mathtt{mv}\,\mathtt{ra},\ell_{\mathbf{2ret}}[\zeta,\epsilon];\\ &\qquad\mathtt{call}\,\ell_2\,\{\forall[].\{\mathtt{r1}\colon\mathbf{int};\zeta\}^{\epsilon}::\zeta,0\}\\ &\mathbf{H}(\ell_{\mathbf{1ret}})=\mathbf{code}[]\{\mathtt{r1}\colon\mathbf{int};\bullet\}^{\mathbf{end}\{\mathbf{int};\bullet\}}.\\ &\qquad\mathtt{halt}\,\mathbf{int},\bullet\,\{\mathtt{r1}\}\\ &\mathbf{H}(\ell_2)=\mathbf{code}[\zeta,\epsilon]\{\mathtt{ra}\colon\forall[].\{\mathtt{r1}\colon\mathbf{int};\zeta\}^{\epsilon};\zeta\}^{\mathtt{ra}}.\\ &\qquad\mathtt{mv}\,\mathtt{r1},1;\mathtt{jmp}\,\ell_{\mathbf{2aux}}[\zeta,\epsilon]\\ &\mathbf{H}(\ell_{\mathbf{2aux}})=\mathbf{code}[\zeta,\epsilon]\{\mathtt{r1}\colon\mathbf{int},\mathtt{ra}\colon\forall[].\{\mathtt{r1}\colon\mathbf{int};\zeta\}^{\epsilon};\zeta\}^{\mathtt{ra}}.\\ &\qquad\mathtt{mult}\,\mathtt{r1},\mathtt{r1},2;\mathtt{ret}\,\mathtt{ra}\,\{\mathtt{r1}\}\\ &\mathbf{H}(\ell_{\mathbf{2ret}})=\mathbf{code}[\zeta,\epsilon]\{\mathtt{r1}\colon\mathbf{int};\forall[].\{\mathtt{r1}\colon\mathbf{int};\zeta\}^{\epsilon}::\zeta\}^{0}.\\ &\qquad\mathtt{sld}\,\mathtt{ra},0;\mathtt{sfree}\,1;\mathtt{ret}\,\mathtt{ra}\,\{\mathtt{r1}\}\end{aligned}$$

Figure 3. $\mathbb{T}$ Example: Call to Call

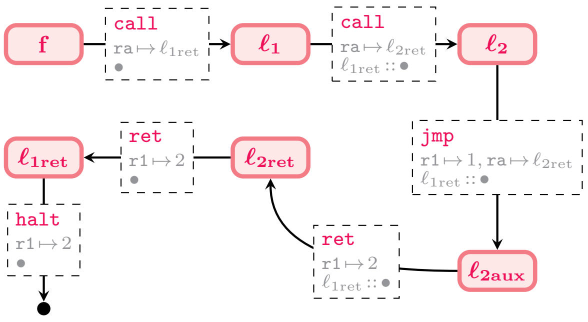


Figure 4. $\mathbb{T}$ Control Flow: Call to Call (Fig. 3)

Example In Figure 3, we show an example $\mathbb{T}$ program demonstrating `call`, `jmp`, `ret`, and `halt`. The control flow, in Figure 4, shows the instructions causing jumps between basic blocks and the state of the relevant registers and stack at jump-time. In this diagram, ℓ_2 and $\ell_{\mathbf{2aux}}$ are in the same component, while the rest are made up of distinct components that together make up the component $\mathbf{f}$.

4. $\mathbb{FT}$ Multi-Language

We present a minimal functional language $\mathbb{F}$ and then embed $\mathbb{F}$ and $\mathbb{T}$ within a Matthews-Findler style multi-language. Particularly notable are the boundary translations for higher-order functions and code blocks. In §5, we design a logical relation with which we can show equivalence of programs that differ both structurally and algorithmically.

Type τ	::=	$\alpha \mid \mathsf{unit} \mid \mathsf{int} \mid (\overline{\tau}) \to \tau \mid \mu\alpha.\tau \mid \langle\overline{\tau}\rangle$
Expression e	::=	$\mathsf{x} \mid () \mid \mathsf{n} \mid \mathsf{e}\ \mathsf{p}\ \mathsf{e} \mid \mathsf{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e} \mid \lambda(\overline{\mathsf{x}:\tau}).\mathsf{e} \mid \mathsf{e}\ \overline{\mathsf{e}}$
		$\mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{e} \mid \mathsf{unfold}\ \mathsf{e} \mid \langle\overline{\mathsf{e}}\rangle \mid \pi_{\mathsf{i}}(\mathsf{e})$
		where $\mathsf{p} ::= + \mid - \mid *$
Value v	::=	$() \mid \mathsf{n} \mid \lambda(\overline{\mathsf{x}:\tau}).\mathsf{e} \mid \mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{v} \mid \langle\overline{\mathsf{v}}\rangle$
Evaluation ctxt E	::=	$[\cdot] \mid \mathsf{E}\ \mathsf{p}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{p}\ \mathsf{E} \mid \mathsf{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e} \mid \mathsf{E}\ \overline{\mathsf{e}} \mid \mathsf{v}\ \overline{\mathsf{v}}\ \mathsf{E}\ \overline{\mathsf{e}}$
		$\mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{E} \mid \mathsf{unfold}\ \mathsf{E} \mid \langle\overline{\mathsf{v}}, \mathsf{E}, \overline{\mathsf{e}}\rangle \mid \pi_{\mathsf{i}}(\mathsf{E})$

Figure 5. $\mathbb{F}$ Syntax

4.1 Functional Language: $\mathbb{F}$

In Figure 5 we present the syntax of $\mathbb{F}$, our simply-typed call-by-value functional language with iso-recursive types, conditional branching, tuples, and base value integers and unit. The language is featureful enough to implement simple programs, while lacking certain expressiveness (like mutation) that we can add by way of the embedded assembly. The typing and operational semantics are standard and provided in the technical appendix [21].

4.2 Embedding $\mathbb{T}$ in $\mathbb{FT}$

Syntax In Figure 6 we present the syntax of our multi-language $\mathbb{FT}$, which is largely made up of extensions to syntactic categories of either $\mathbb{T}$ (Figure 1) or $\mathbb{F}$ (Figure 5). Note that both expressions e and components $\mathbf{e}$ are components e in this language. Henceforth, when we refer to an $\mathbb{F}$ or $\mathbb{T}$ term we are referring to the terms that originated in that language, which can now of course include nested components of the other language. We add boundaries ${}^{\tau}\mathcal{FT}\ \mathbf{e}$ ($\mathbb{T}$ inside, $\mathbb{F}$ outside) and $\mathcal{TF}^{\tau}\ \mathsf{e}$ ($\mathbb{F}$ inside, $\mathbb{T}$ outside) to mediate between the languages. In both cases, the $\mathbb{F}$ type τ directs the translation. In particular, the ${}^{\tau}\mathcal{FT}\ \mathbf{e}$ contains a $\mathbb{T}$ component $\mathbf{e}$ with $\mathbb{T}$ translated type $\tau^{\mathcal{T}}$, while the $\mathcal{TF}^{\tau}\ \mathsf{e}$ contains an $\mathbb{F}$ expression e of type τ. Like Matthews-Findler [16], we reduce the component within the boundary to a value, after which we carry out a type-directed value translation using translation metafunctions ${}^{\tau}\mathbf{FT}(\cdot)$ and $\mathbf{TF}^{\tau}(\cdot)$, e.g.:

$$ {}^{\tau}\mathcal{FT}\ \mathbf{e} \longmapsto^{*} {}^{\tau}\mathcal{FT}\ \mathbf{v} \longmapsto {}^{\tau}\mathbf{FT}(\mathbf{v}) $$

To $\mathbb{T}$ instructions ι, we add an `import` instruction to wrap the boundary and to specify what register the translated value should be placed in. The `import` instruction also specifies σ, the tail of the stack that should be protected while evaluating the $\mathbb{F}$ expression e, which could in turn include $\mathbb{T}$ code. Consider the following concrete example, which computes the $\mathbb{F}$ expression $\mathbf{1} + \mathbf{1}$ and loads it into register $\mathtt{r1}$, protecting the whole stack—here, just the empty stack—while doing it:

Type τ	::=	$\cdots \mid (\overline{\tau}) \xrightarrow{\phi;\phi} \tau'$
Expression e	::=	$\cdots \mid {}^{\tau}\mathcal{FT}\ \mathbf{e} \mid \lambda^{\phi}_{\phi}(\overline{\mathsf{x}:\tau}).\mathsf{t} \mid \mathsf{t}\ \overline{\mathsf{t}'}$
Return marker $\mathbf{q}$	::=	$\cdots \mid \mathbf{out}$
Instruction sequence $\mathbf{I}$	::=	$\cdots \mid \mathtt{protect}\ \phi, \zeta; \mathbf{I}$
Instruction ι	::=	$\cdots \mid \mathtt{import}\ \mathtt{r_d}, {}^{\sigma}\mathcal{TF}^{\tau}\ \mathsf{e}$
Stack prefix ϕ	::=	$\cdot \mid \tau :: \phi$
Stack typing σ	::=	$\phi :: \zeta \mid \phi :: \bullet$
Evaluation ctxt E	::=	$\cdots \mid {}^{\tau}\mathcal{FT}\ \mathbf{E}$
Evaluation ctxt $\mathbf{E}$	::=	$\cdots \mid (\mathtt{import}\ \mathtt{r_d}, {}^{\sigma}\mathcal{TF}^{\tau}\ \mathsf{E}; \mathbf{I}, \cdot)$
Type τ	::=	$\tau \mid \tau$
Component e	::=	$\mathsf{e} \mid \mathbf{e}$
Δ	::=	$\cdot \mid \Delta, \alpha \mid \Delta, \alpha \mid \Delta, \zeta \mid \Delta, \epsilon$
Evaluation ctxt E	::=	$\mathsf{E} \mid \mathbf{E}$

Figure 6. $\mathbb{FT}$ Multi-Language Syntax

$$ \cdot\,;\ \cdot\,;\ \cdot\,;\ \bullet\,;\ \mathbf{end}\{\mathbf{int};\bullet\} \vdash \mathtt{import}\ \mathtt{r1}, {}^{\bullet}\mathcal{TF}^{\mathsf{int}}\ (\mathbf{1}+\mathbf{1}) $$
$$ \Rightarrow\ \cdot\ ; \mathtt{r1}:\mathbf{int}\ ; \bullet\ ;\ \mathbf{end}\{\mathbf{int};\bullet\} $$

When translating $\mathbb{T}$ code blocks into $\mathbb{F}$ functions, we will need to instantiate the stack tail variable ζ on the $\mathbb{T}$ code block. For this reason, we introduce the `protect` instruction, which specifies a stack prefix ϕ to leave visible and a type variable ζ to bind to the tail. We will see the value translation later in the section.

While normal $\mathbb{F}$ lambdas are embedded in the multi-language, they do not allow stack modification in embedded $\mathbb{T}$ code. However, we may want to allow that sort of modification. For this reason, we introduce an optional new stack-modifying lambda term $\lambda^{\phi_{\mathrm{i}}}_{\phi_{\mathrm{o}}}(\overline{\mathsf{x}:\tau}).\mathsf{e}$, which specifies the stack prefix ϕ_{i} it requires on the front of the stack when it is called, and the stack prefix ϕ_{o} that it will have replaced ϕ_{i} with upon return. Correspondingly, we introduce a new arrow type that captures that relationship. Note that the ordinary lambda can be seen as a special case when ϕ_{i} and ϕ_{o} are both the empty prefix $\cdot$, which corresponds to the entire stack being the protected tail. While there is no fundamental reason these stack-modifying lambdas must be included, we can use them, for instance, to write a function that pushes the number $\mathbf{7}$ onto the stack using embedded assembly:

$$ \lambda^{\bullet}_{\mathbf{int}\,::\,\bullet}(\overline{\mathsf{x}:\mathsf{int}}).{}^{\mathsf{unit}}\mathcal{FT}\ (\mathtt{protect}\ \cdot, \zeta; \mathtt{mv}\ \mathtt{r1}, \mathbf{7}; \mathtt{salloc}\ \mathbf{1}; $$
$$ \mathtt{sst}\ \mathbf{0}, \mathtt{r1}; \mathtt{mv}\ \mathtt{r1}, (); $$
$$ \mathtt{halt}\ \mathbf{unit}, \mathbf{int} :: \zeta\ \{\mathtt{r1}\}, \cdot) $$

The inline assembly of this function first captures the current stack as an abstract ζ, then loads $\mathbf{7}$ into register $\mathtt{r1}$, allocates a cell on the stack and stores the value there, before clearing out $\mathtt{r1}$ and halting on it. Without stack-modifying lambdas, this would fail to type check since the stack at

$$\boxed{\Psi; \Delta; \Gamma; \chi; \sigma; q \vdash e : \tau; \sigma'}$$

$$\frac{\Psi; \Delta; \Gamma; \chi; \sigma; \mathrm{out} \vdash t : (\tau_1 \cdots \tau_n) \rightarrow \tau'; \sigma_0 \qquad \Psi; \Delta; \Gamma; \chi; \sigma_{i-1}; \mathrm{out} \vdash t_i : \tau_i; \sigma_i}{\Psi; \Delta; \Gamma; \chi; \sigma; \mathrm{out} \vdash t\, t_1 \cdots t_n : \tau'; \sigma_n}$$

$$\frac{\Psi; \Delta; \Gamma; \cdot; \sigma; \mathrm{end}\{\tau^{\mathcal{T}}; \sigma'\} \vdash e : \tau^{\mathcal{T}}; \sigma'}{\Psi; \Delta; \Gamma; \chi; \sigma; \mathrm{out} \vdash {}^{\tau}\mathcal{FT}\, e : \tau; \sigma'}$$

$$\frac{\Psi; \Delta, \zeta; \Gamma, \overline{x : \tau}; \chi; \phi_i :: \zeta; \mathrm{out} \vdash t : \tau'; \phi_o :: \zeta}{\Psi; \Delta; \Gamma; \chi; \sigma; \mathrm{out} \vdash \lambda^{\phi_i}_{\phi_o}(\overline{x : \tau}).t : (\overline{\tau}) \xrightarrow{\phi_i;\phi_o} \tau'; \sigma}$$

$$\boxed{\Psi; \Delta; \Gamma; \chi; \sigma; q \vdash I} \text{ where } \cdot[\Delta]; \chi; \sigma \vdash q$$

$$\frac{\sigma = \phi :: \sigma_0 \qquad \sigma' = \phi :: \zeta \qquad \Psi; \Delta, \zeta; \Gamma; \chi; \sigma'; q \vdash I}{\Psi; \Delta; \Gamma; \chi; \sigma; q \vdash \mathtt{protect}\ \phi, \zeta; I}$$

$$\boxed{\Psi; \Delta; \Gamma; \chi; \sigma; q \vdash \iota \Rightarrow \Delta'; \chi'; \sigma'; q'} \text{ where } \cdot[\Delta]; \chi; \sigma \vdash q$$

$$\frac{\begin{array}{cc} \sigma = \tau_0 :: \cdots :: \tau_j :: \sigma_0 & \sigma' = \tau'_0 :: \cdots :: \tau'_k :: \sigma_0 \\ \sigma^* = \tau_0 :: \cdots :: \tau_j :: \zeta & \sigma'^* = \tau'_0 :: \cdots :: \tau'_k :: \zeta \end{array} \\ \Psi; \Delta, \zeta; \Gamma; \chi; \sigma^*; \mathrm{out} \vdash e : \tau; \sigma'^* \\ q = i > j \text{ or } q = \mathrm{end}\{\hat{\tau}; \hat{\sigma}\}}{\Psi; \Delta; \Gamma; \chi; \sigma; q \vdash \mathtt{import}\ r_d, {}^{\sigma_0}\mathcal{TF}^{\tau}\, e \\ \qquad \Rightarrow \Delta; (r_d : \tau^{\mathcal{T}}); \sigma'; \mathrm{inc}(q, k{-}j)}$$

Figure 7. Selected $\mathbb{FT}$ Typing Rules

the end of the body of the lambda would be different than it had been at the beginning. In our technical appendix and artifact we use this feature to implement a very basic mutable reference library.

Finally, in Figure 6 we also add a new return marker **out**, which is used for $\mathbb{F}$ code, since $\mathbb{F}$ follows normal expression-based evaluation and thus has no return continuation.

Type System The typing judgments for $\mathbb{FT}$, for which we show a selection in Figure 7, include modified versions from both $\mathbb{T}$ and $\mathbb{F}$ judgments as well as rules for the new forms. Since this is a multi-language and not a compiler, the typing rules for $\mathbb{T}$ must now include an $\mathbb{F}$ environment Γ of free $\mathbb{F}$ variables. Similarly, the typing rules for $\mathbb{F}$ must now include all of the context needed by $\mathbb{T}$, since in order to type-check embedded assembly components we will need to know the current register (χ), stack (σ), and heap (Ψ) typings.

Most of these modifications are straightforward; we show the rule for $\mathbb{F}$ application in Figure 7 as a representative. Note that the stack typings σ_i are threaded through the arguments according to evaluation order, as each one could include embedded $\mathbb{T}$ code that modified the stack.

For the boundary term, ${}^{\tau}\mathcal{FT}\ e$, we require that the $\mathbb{T}$ component e within the boundary be well typed under translation type $\tau^{\mathcal{T}}$ and return marker $\mathrm{end}\{\tau^{\mathcal{T}}; \sigma'\}$, which corresponds to the inner assembly halting with a value of type $\tau^{\mathcal{T}}$. In that case, the boundary term is well typed under τ at the **out** return marker that corresponds to $\mathbb{F}$ code. Note that the boundary makes no restriction on modification of the stack. Also in the figure is the typing rule for the stack-modifying lambda term, which is an ordinary lambda typing rule except it types under stacks with the given prefixes ϕ_i and ϕ_o and abstract tails ζ; as noted before, the regular lambda is a special case when ϕ_i and ϕ_o are empty.

$$\langle M \mid E[{}^{\tau}\mathcal{FT}\,(\mathtt{halt}\ \tau^{\mathcal{T}}, \sigma\ \{\mathtt{r}\}, \cdot)] \rangle$$
$$\longmapsto \langle M' \mid E[v] \rangle \qquad \text{if } {}^{\tau}\mathbf{FT}(M.R(\mathtt{r}), M) = (v, M')$$
$$\langle M \mid E[\mathtt{import}\ r_d, {}^{\sigma'}\mathcal{TF}^{\tau}\ v; I] \rangle$$
$$\longmapsto \langle M' \mid E[\mathtt{mv}\ r_d, w; I] \rangle \qquad \text{if } \mathbf{TF}^{\tau}(v, M) = (w, M')$$

Figure 8. $\mathbb{FT}$ Operational Semantics: Language Boundaries

$$\begin{aligned} \alpha^{\mathcal{T}} &= \alpha \\ \mathsf{unit}^{\mathcal{T}} &= \mathrm{unit} \qquad\qquad \mu\alpha.\tau^{\mathcal{T}} = \mu\alpha.(\tau^{\mathcal{T}}) \\ \mathsf{int}^{\mathcal{T}} &= \mathrm{int} \qquad \langle \tau_1, \ldots, \tau_n \rangle^{\mathcal{T}} = \mathrm{box}\ \langle \tau_1^{\mathcal{T}}, \ldots, \tau_n^{\mathcal{T}} \rangle \end{aligned}$$
$$(\tau_1, \ldots, \tau_n) \rightarrow \tau'^{\mathcal{T}} =$$
$$\mathrm{box}\ \forall[\zeta, \epsilon].\{\mathtt{ra} : \mathrm{box}\ \forall[].\{\mathtt{r1} : \tau'^{\mathcal{T}}; \zeta\}^{\epsilon}; \sigma'\}^{\mathtt{ra}}$$
$$\text{where } \sigma' = \tau_n^{\mathcal{T}} :: \cdots :: \tau_1^{\mathcal{T}} :: \zeta$$
$$(\tau_1, \ldots, \tau_n) \xrightarrow{\phi_i;\phi_o} \tau'^{\mathcal{T}} =$$
$$\mathrm{box}\ \forall[\zeta, \epsilon].\{\mathtt{ra} : \mathrm{box}\ \forall[].\{\mathtt{r1} : \tau'^{\mathcal{T}}; \phi_o :: \zeta\}^{\epsilon}; \sigma'\}^{\mathtt{ra}}$$
$$\text{where } \sigma' = \tau_n^{\mathcal{T}} :: \cdots :: \tau_1^{\mathcal{T}} :: \phi_i :: \zeta$$

Figure 9. $\mathbb{FT}$ Boundary Type Translation

As described above, we add two new $\mathbb{T}$ instructions. The `protect` instruction is used to abstract the tail of the stack, which we can see in the transformation of the stack $\phi :: \sigma_0$ into $\phi :: \zeta$ when typing the subsequent instruction sequence I, where ζ is a new type variable introduced to the type environment. Note that there is no way to undo this; it lasts until the end of the current $\mathbb{T}$ component. If q is i, `protect` should not be allowed to hide the ith stack slot in ζ; this is enforced by the restrictions on q (see §3) when typing I.

The other new instruction is the $\mathbb{T}$ boundary instruction `import`. Ignoring stacks, the rule is quite simple: it takes an $\mathbb{F}$ term e of type τ, well typed under the **out** return marker, and translates it to type $\tau^{\mathcal{T}}$, storing the result in register r_d. This story is complicated by the handling of stacks, as it is important for `import` instructions to be able to restrict what portion of the stack the inner code can modify. In particular, since the $\mathbb{F}$ code does not have the same return marker q, we must be sure that q cannot be clobbered by $\mathbb{T}$ code embedded in e. To do this, we specify the portion of the stack σ_0 that is abstracted as ζ in e, and ensure that either q is stored in that stack tail or it is the halting marker. Finally, since the front of the stack could grow or shrink to k entries, if q were a stack index i we increment it by $k - j$ using the metafunction inc, which otherwise is identity.

$$
\begin{aligned}
&\mathbf{TF}^{\mathsf{int}}(\mathsf{n}, \mathrm{M}) = (\mathbf{n}, \mathrm{M})\\
&\mathbf{TF}^{\mu\alpha.\tau}(\mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{v}, \mathrm{M}) = (\mathbf{fold}_{\mu\alpha.\tau^{\mathcal{T}}}\ \mathrm{v}, \mathrm{M}')\\
&\qquad \text{where } \mathbf{TF}^{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{v}, \mathrm{M}) = (\mathrm{v}, \mathrm{M}')\\
&\mathbf{TF}^{\langle\tau_1, \ldots, \tau_n\rangle}(\langle \mathsf{v}_0, \ldots, \mathsf{v}_n\rangle, \mathrm{M}) =\\
&\qquad (\ell, (\mathrm{M}_{n+1}, \ell \mapsto \langle \mathrm{w}_0, \ldots, \mathrm{w}_n\rangle))\\
&\qquad \text{where } \mathrm{M}_0 = \mathrm{M}, \text{ and } \mathbf{TF}^{\tau_i}(\mathsf{v}_i, \mathrm{M}_i) = (\mathrm{w}_i, \mathrm{M}_{i+1})\\
&\mathbf{TF}^{\mathsf{unit}}((), \mathrm{M}) = ((), \mathrm{M})\\
&\mathbf{TF}^{(\overline{\tau}) \to \tau'}(\lambda(\overline{\mathsf{x} : \tau}).\mathsf{t}, \mathrm{M}) = (\ell, (\mathrm{M}, \ell \mapsto \mathbf{h}))\\
&\text{where } \mathbf{h} = \mathbf{code}[\zeta, \epsilon]\{\mathtt{ra} : \forall[].\{\mathtt{r1} : \tau'^{\mathcal{T}}; \zeta\}^{\epsilon}; \overline{\tau^{\mathcal{T}}} :: \zeta\}^{\mathtt{ra}}.\\
&\qquad\qquad \mathtt{salloc}\ 1; \mathtt{sst}\ 0, \mathtt{ra}; \mathtt{import}\ \mathrm{r}_1, {}^{\zeta}\mathcal{TF}^{\tau'}\ \mathsf{e};\\
&\qquad\qquad \mathtt{sld}\,\mathtt{ra}, 0; \mathtt{sfree}\ \mathrm{n}{+}1; \mathtt{ret}\ \mathtt{ra}\ \{\mathtt{r1}\}\\
&\qquad \mathsf{e} = (\lambda(\overline{\mathsf{x} : \tau}).\mathsf{t})^{\overline{\tau}\mathcal{FT}}\ (\overline{\mathtt{sld}\,\mathtt{r1}, \mathrm{n}{+}1{-}\mathrm{i};}\\
&\qquad\qquad\qquad \overline{\mathtt{halt}\ \tau^{\mathcal{T}}, \sigma\ \{\mathtt{r1}\}, \cdot)}\\
&\qquad \sigma = \forall[].\{\mathtt{r1} : \tau'^{\mathcal{T}}; \zeta\}^{\epsilon} :: \overline{\tau^{\mathcal{T}}} :: \zeta\\
&{}^{\mathsf{unit}}\mathbf{FT}((), \mathrm{M}) = ((), \mathrm{M})\\
&{}^{\mathsf{int}}\mathbf{FT}(\mathrm{n}, \mathrm{M}) = (\mathsf{n}, \mathrm{M})\\
&{}^{\mu\alpha.\tau}\mathbf{FT}(\mathrm{fold}_{\mu\alpha.\tau^{\mathcal{T}}}\ \mathrm{w}) = (\mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{v}, \mathrm{M}')\\
&\qquad \text{where } {}^{\tau[\mu\alpha.\tau/\alpha]}\mathbf{FT}(\mathrm{w}, \mathrm{M}) = (\mathsf{v}, \mathrm{M}')\\
&{}^{\langle\tau_0, \ldots, \tau_n\rangle}\mathbf{FT}(\ell, \mathrm{M}) = (\langle \mathsf{v}_0, \ldots, \mathsf{v}_n\rangle, \mathrm{M}_{n+1})\\
&\qquad \text{where } \mathrm{M}(\ell) = \langle \mathrm{w}_0, \ldots, \mathrm{w}_n\rangle,\\
&\qquad\qquad \mathrm{M}_0 = \mathrm{M}, \text{ and } {}^{\tau_i}\mathbf{FT}(\mathrm{w}_i, \mathrm{M}_i) = (\mathsf{v}_i, \mathrm{M}_{i+1})\\
&{}^{(\overline{\tau_n}) \to \tau'}\mathbf{FT}(\mathrm{w}, \mathrm{M}) = (\mathsf{v}, (\mathrm{M}, \ell_{\mathrm{end}} \mapsto \mathrm{h}_{\mathrm{end}}))\\
&\text{where } \mathsf{v} = \lambda(\overline{\mathsf{x}_n : \tau_n}).{}^{\tau'}\mathcal{FT}\ (\mathtt{protect}\ \cdot, \zeta;\\
&\qquad\qquad \mathtt{import}\ \mathtt{r1}, {}^{\zeta}\mathcal{TF}^{\tau_1}\ \mathsf{x}_1; \mathtt{salloc}\ 1; \mathtt{sst}\ 0, \mathtt{r1}; \ldots\\
&\qquad\qquad \mathtt{import}\ \mathtt{r1}, {}^{\zeta}\mathcal{TF}^{\tau_n}\ \mathsf{x}_n; \mathtt{salloc}\ 1; \mathtt{sst}\ 0, \mathtt{r1};\\
&\qquad\qquad \mathtt{mv}\ \mathtt{ra}, \ell_{\mathrm{end}}[\zeta]; \mathtt{call}\ \mathrm{w}\ \{\zeta, \mathrm{end}\{\tau'^{\mathcal{T}}; \zeta\}\}, \cdot)\\
&\mathrm{h}_{\mathrm{end}} = \mathrm{code}[\zeta]\{\mathtt{r1} : \tau'^{\mathcal{T}}; \zeta\}^{\mathrm{end}\{\tau'^{\mathcal{T}}; \zeta\}}.\\
&\qquad\qquad \mathtt{halt}\ \tau'^{\mathcal{T}}, \zeta\ \{\mathtt{r1}\}
\end{aligned}
$$

Figure 10. $\mathbb{FT}$ Boundary Value Translation

Operational Semantics The operational semantics for boundary terms, shown in Figure 8, translate values using the type-directed metafunctions ${}^{\tau}\mathbf{FT}(\cdot)$ ($\mathbb{T}$ inside, $\mathbb{F}$ outside) and $\mathbf{TF}^{\tau}(\cdot)$ ($\mathbb{F}$ inside, $\mathbb{T}$ outside).

Figure 9 contains the type translation guiding these metafunctions. Note that $\mathbb{F}$ tuples are translated to immutable references to $\mathbb{T}$ heap tuples. The most complex transformation is for function types, which are translated into code blocks that pass arguments on the stack and follow the calling convention described in §3 where return continuations can be instantiated alternately by $\mathbb{T}$ or $\mathbb{F}$ callers.

We show the value translations in Figure 10, eliding only the stack-modifying lambda, which is similar to the lambda shown. The most significant translations are between $\mathbb{T}$ code blocks and $\mathbb{F}$ functions. In particular, we must translate between variable representations and calling conventions—this means the arguments are passed on the stack, and a return continuation must be in register `ra`. Finally, we must translate the arguments themselves, and translate the return value back, cleaning up temporary stack values.

Critically, when translating an $\mathbb{F}$ function to a $\mathbb{T}$ code block, we must protect the return continuation, since embedded assembly blocks within the body of the function could write to register `ra`. To do that, we store `ra`'s contents on the stack and protect the tail. In the stack-modifying lambda case, this is complicated slightly by needing to re-arrange the stack to put the protected value past the exposed stack prefix ϕ_i. To evaluate the $\mathbb{F}$ function, we load each argument from the stack, translate it to $\mathbb{F}$, apply the function, and import the returned value back to $\mathbb{T}$. After doing this, we load the return continuation off of the stack, clear the arguments according to the calling convention and return. Note that in the stack-modifying lambda case, we have to be careful to clear the arguments but keep the output prefix ϕ_o.

Example In Figure 11, we present an example of the type of transformation that a JIT compiler could perform and the resulting higher-order callbacks that appear in the multi-language program. At the top of the figure is the $\mathbb{F}$ source program which has three functions: **g** passes **1** to its argument, **h** doubles its argument, and **f** passes **h** to its argument. The functions themselves are intentionally minimal, but we assume the JIT compiler determined that **f** and **h** should be compiled to assembly and present the transformed program in the lower half of the figure. Here, **f** and **h** have been replaced by code blocks pointed to by ℓ and ℓ_h respectively.

We present a control-flow diagram for the transformed program in Figure 12, where arrows in $\mathbb{F}$ boxes correspond to argument passing and return values, whereas arrows in $\mathbb{T}$ boxes correspond to jumps or halt (as in Figure 4).

In this example, when control passes to ℓ, which was compiled from **f**, we need to be able to call back into the high-level code in **g**. In the block pointed to by ℓ, according to the calling convention, the argument **g** is passed on the top of the stack. This means that to call back to it, we load it off the stack into register `r1` with instruction `sld r1, 0`, and then `call` it, as shown in the control-flow diagram in the transfer from box $\boxed{\ell}$ to box $\boxed{\mathbf{g}}$.

But in this example, and indeed in any JIT for higher-order languages, we may not only need to call from compiled assembly to the interpreted language, but also be able to pass compiled code back as arguments to the interpreted language. In this example, the ℓ_h component, which was compiled from **h**, is passed as an argument to **g**. The function **g** then calls ℓ_h with **1**, causing control to transfer back to ℓ_h as we can see in the transfer to the innermost block in the control-flow diagram.

The value translation (shown in Figure 10) introduces extra blocks where needed, colored as $\boxed{\ell_{\mathrm{hret}}}$ and $\boxed{\ell_{\mathrm{ret}}}$ in our diagram. These are needed because $\mathbb{T}$ components jump to continuation blocks, whereas for control to pass back to $\mathbb{F}$ they must `halt`, which these shim-blocks achieve.

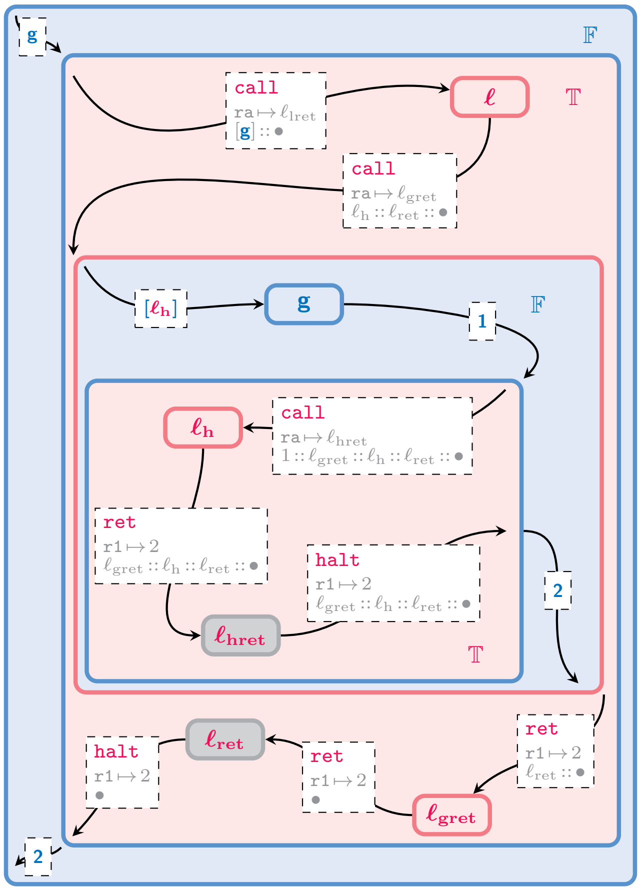
Even though small, this example demonstrates how mixed-language programs with higher-order callbacks arise naturally in the context of JIT compilation. In the next section, we'll see how we can use our logical relation to prove these types of programs equivalent, a necessary step for any proof of correctness for a JIT compiler.

$$\begin{aligned}
&\mathsf{g} = \lambda(\mathsf{h}\colon(\mathsf{int})\to\mathsf{int}).\mathsf{h}\,1\\
&\mathsf{h} = \lambda(\mathsf{x}\colon\mathsf{int}).\mathsf{x} * 2\\
&\mathsf{f} = \lambda(\mathsf{g}\colon((\mathsf{int})\to\mathsf{int})\to\mathsf{int}).\mathsf{g}\,\mathsf{h}\\
&\mathsf{e} = \mathsf{f}\,\mathsf{g}
\end{aligned}$$

↓ JIT Compile ↓

$$\begin{aligned}
&\tau = ((\mathsf{int})\to\mathsf{int})\to\mathsf{int}\\
&\mathsf{g} = \lambda(\mathsf{h}\colon(\mathsf{int})\to\mathsf{int}).\mathsf{h}\,1\\
&\mathsf{e} = ({}^{\mathsf{int}}\mathcal{FT}\,(\mathtt{mv\,r1},\ell;\mathtt{halt}\,(\tau)\to\mathsf{int}^{\mathcal{T}},\bullet\,\{\mathtt{r1}\},\mathbf{H}))\,\mathsf{g}\\
&\mathbf{H}(\ell) = \mathbf{code}[\zeta,\epsilon]\{\mathtt{ra}\colon\forall[].\{\mathtt{r1}\colon\mathsf{int}^{\mathcal{T}};\zeta\}^{\epsilon};\tau^{\mathcal{T}} :: \zeta\}^{\mathtt{ra}}.\\
&\qquad\qquad \mathtt{sld\,r1},0;\mathtt{salloc}\,1;\mathtt{mv\,r2},\ell_{\mathrm{h}};\mathtt{sst}\,0,\mathtt{r2};\\
&\qquad\qquad \mathtt{sst}\,1,\mathtt{ra};\mathtt{mv\,ra},\ell_{\mathrm{gret}}[\zeta,\epsilon];\\
&\qquad\qquad \mathtt{call\,r1}\,\{\forall[].\{\mathtt{r1}\colon\mathsf{int}^{\mathcal{T}};\zeta\}^{\epsilon} :: \zeta,0\}\\
&\mathbf{H}(\ell_{\mathrm{h}}) = \mathbf{code}[\zeta,\epsilon]\{\mathtt{ra}\colon\forall[].\{\mathtt{r1}\colon\mathsf{int}^{\mathcal{T}};\zeta\}^{\epsilon};\mathsf{int}^{\mathcal{T}} :: \zeta\}^{\mathtt{ra}}.\\
&\qquad\qquad \mathtt{sld\,r1},0;\mathtt{sfree}\,1;\mathtt{mul\,r1},\mathtt{r1},2;\mathtt{ret\,ra}\,\{\mathtt{r1}\}\\
&\mathbf{H}(\ell_{\mathrm{gret}}) = \mathbf{code}[\zeta,\epsilon]\{\mathtt{r1}\colon\mathtt{int};\forall[].\{\mathtt{r1}\colon\mathsf{int}^{\mathcal{T}};\zeta\}^{\epsilon} :: \zeta\}^{0}.\\
&\qquad\qquad \mathtt{sld\,ra},0;\mathtt{sfree}\,1;\mathtt{ret\,ra}\,\{\mathtt{r1}\}
\end{aligned}$$

Figure 11. $\mathbb{FT}$ Example: JIT



g
$\mathbb{F}$
call
ra ↦ ℓ_{ret}
[g] :: •
ℓ
$\mathbb{T}$
call
ra ↦ ℓ_{gret}
ℓ_{h} :: ℓ_{ret} :: •
[ℓ_{h}]
g
1
$\mathbb{F}$
ℓ_{h}
call
ra ↦ ℓ_{hret}
1 :: ℓ_{gret} :: ℓ_{h} :: ℓ_{ret} :: •
ret
r1 ↦ 2
ℓ_{gret} :: ℓ_{h} :: ℓ_{ret} :: •
halt
r1 ↦ 2
ℓ_{gret} :: ℓ_{h} :: ℓ_{ret} :: •
2
ℓ_{hret}
$\mathbb{T}$
ret
r1 ↦ 2
ℓ_{ret} :: •
halt
r1 ↦ 2
•
ℓ_{ret}
ret
r1 ↦ 2
•
ℓ_{gret}
2

Figure 12. $\mathbb{FT}$ Control Flow: JIT (Fig. 11)

5. Logical Relation for $\mathbb{FT}$

In order to reason about program equivalence in $\mathbb{FT}$, we design a step-indexed Kripke logical relation for our language. Our logical relation builds on that of Dreyer *et al.* [10] and Ahmed *et al.* [4], where the Kripke worlds contain *islands* with state-transition systems that we use to accommodate mutations to the heap, registers, and stack. From those models, we inherit the ability to reason about equivalences dependent on hidden mutable state, though we won't go into detail about that aspect in this paper. In this section, we focus on the novel aspects of our logical relation, showing how we adapted the earlier models to the setting of $\mathbb{FT}$. In particular, the addition of return markers required non-trivial extensions to the model.

In our logical relation, for which we show the closed relations in Figure 13, we have three value relations: $\mathcal{V}[\![\tau]\!]\rho$, $\mathcal{W}[\![\tau]\!]\rho$, and $\mathcal{HV}[\![\psi]\!]\rho$. These correspond to the three types of values that exist in $\mathbb{FT}$: high-level values, low-level word-sized values, and low-level heap values, respectively. As usual in these relations, ρ is a relational substitution for type variables. Further, with the exception of contexts in the $\mathcal{K}$ relation all of our relations are built out of well-typed terms, though we elide that requirement in these figures.

In a Kripke logical relation, relatedness of values depends on the state of a world W. Some values are related irrespective of world state; for example, an integer $\mathbf{n}$ is related to itself in any world W, written $(W, \mathbf{n}, \mathbf{n}) \in \mathcal{W}[\![\mathbf{int}]\!]\rho$. However, the structure of the world captures key semantic properties about the stack, heap, and registers in a sequence of *islands* that describe the current state of memories. Each island expresses invariants on certain parts of memory by encoding a state-transition system and a memory relation that establishes which pairs of memories are related in each state.

Since our logical relation is step-indexed our worlds have an index k, which conveys that the relation captures semantic equivalence of terms for up to k steps but no information is known beyond that. This allows us to avoid circularity when dealing with recursive types as we can induct on the step index rather than the structure of the expanding type.

$W' \sqsupseteq W$ when W' is a future world of W; to reach it, we may have consumed steps (lowering k), allocated additional memory in new islands, or made transitions in islands.

A novel aspect of our logical relation is how it formalizes equivalence of code blocks at code-pointer type (Figure 15). Our code-pointer logical relation is like a function logical relation in that, given related inputs, it should produce related outputs. Inputs, in this case, are registers and the stack for which, in a future world W' with closing type substitution ρ^*, we require that curr-R$(W') \in \mathcal{R}[\![\chi]\!]\rho'$ and curr-S$(W') \in \mathcal{S}[\![\sigma]\!]\rho'$. This means that the current register files and stacks in world W' are related at register file typing

Statement	Meaning
$(W, \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\boldsymbol{\tau}]\!]\rho$	$\mathbf{v_1}$ and $\mathbf{v_2}$ are related $\mathbb{F}$ values at type $\boldsymbol{\tau}$ in world W under type substitution ρ
$(W, \mathbf{w_1}, \mathbf{w_2}) \in \mathcal{W}[\![\boldsymbol{\tau}]\!]\rho$	$\mathbf{w_1}$ and $\mathbf{w_2}$ are related $\mathbb{T}$ word values at type $\boldsymbol{\tau}$ in world W under type substitution ρ
$(W, \mathbf{h_1}, \mathbf{h_2}) \in \mathcal{HV}[\![\boldsymbol{\psi}]\!]\rho$	$\mathbf{h_1}$ and $\mathbf{h_2}$ are related $\mathbb{T}$ heap values at type $\boldsymbol{\psi}$ in world W under type substitution ρ
$(W, e_1, e_2) \in \mathcal{O}$	e_1 and e_2 run with memories related at W, either both terminate or are both running after $W.k$ steps
$(W, E_1, E_2) \in \mathcal{K}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma}]\!]\rho$	E_1 and E_2 are related continuations, so given appropriately related values at type τ, they are in $\mathcal{O}$
$(W, e_1, e_2) \in \mathcal{E}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma}]\!]\rho$	e_1 and e_2 are related expressions, so given appropriate related continuations, they are in $\mathcal{O}$

Figure 13. $\mathbb{FT}$ Logical Relation: Closed Values and Terms

$$
\begin{aligned}
\mathcal{E}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma}]\!]\rho &= \{ (W, e_1, e_2) \mid \forall E_1, E_2.\ (W, E_1, E_2) \in \mathcal{K}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma}]\!]\rho \implies (W, E_1[e_1], E_2[e_2]) \in \mathcal{O} \} \\
\mathcal{K}[\![\mathbf{out} \vdash \boldsymbol{\tau}; \boldsymbol{\sigma}]\!]\rho &= \{ (W, E_1, E_2) \mid \forall W', \mathbf{v_1}, \mathbf{v_2}. W' \sqsupseteq_{\text{pub}} W \ \wedge\ (W', \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\boldsymbol{\tau}]\!]\rho \ \wedge\ \text{curr-S}(W') \Subset \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho \\
&\qquad \implies (W', E_1[\mathbf{v_1}], E_2[\mathbf{v_1}]) \in \mathcal{O} \} \\
\mathcal{K}[\![\mathbf{end\{\boldsymbol{\tau}; \boldsymbol{\sigma}\}} \vdash \boldsymbol{\tau}; \boldsymbol{\sigma}]\!]\rho &= \{ (W, E_1, E_2) \mid \forall W', \mathbf{r_1}, \mathbf{r_2}. W' \sqsupseteq_{\text{pub}} W \wedge \\
&\qquad (\triangleright W', W'.\mathbf{R_1}(\mathbf{r_1}), W'.\mathbf{R_2}(\mathbf{r_2})) \in \mathcal{W}[\![\boldsymbol{\tau}]\!]\rho \ \wedge\ \text{curr-S}(W') \Subset \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho \\
&\qquad \implies (W', E_1[(\mathtt{halt}\ \rho_1(\boldsymbol{\tau}), \rho_1(\boldsymbol{\sigma})\ \{\mathbf{r_1}\}, \cdot)], E_2[(\mathtt{halt}\ \rho_2(\boldsymbol{\tau}), \rho_2(\boldsymbol{\sigma})\ \{\mathbf{r_2}\}, \cdot)]) \in \mathcal{O} \} \\
\mathcal{K}[\![\mathbf{q} \vdash \boldsymbol{\tau}; \boldsymbol{\sigma}]\!]\rho &= \{ (W, E_1, E_2) \mid (\mathbf{q} = \mathbf{r} \ \vee\ \mathbf{q} = \mathbf{i}) \ \wedge\ \forall W', \mathbf{q'}, \mathbf{r_1}, \mathbf{r_2}. W' \sqsupseteq_{\text{pub}} W \wedge \\
&\qquad (\exists \mathbf{r}. \mathbf{q'} = \mathbf{r} \ \wedge\ \text{ret-addr}_1(W, \rho_1(\mathbf{q})) = W'.\mathbf{R_1}(\mathbf{r}) \ \wedge\ \text{ret-addr}_2(W, \rho_2(\mathbf{q})) = W'.\mathbf{R_2}(\mathbf{r}) \ \wedge \\
&\qquad\qquad \text{ret-reg}_1(W', \mathbf{r}) = \mathbf{r_1} \ \wedge\ \text{ret-reg}_2(W', \mathbf{r}) = \mathbf{r_2}) \wedge \\
&\qquad (\triangleright W', W'.\mathbf{R_1}(\mathbf{r_1}), W'.\mathbf{R_2}(\mathbf{r_2})) \in \mathcal{W}[\![\boldsymbol{\tau}]\!]\rho \ \wedge\ \text{curr-S}(W') \Subset \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho \\
&\qquad \implies (W', E_1[(\mathtt{ret}\ \rho_1(\mathbf{q'})\ \{\mathbf{r_1}\}, \cdot)], E_2[(\mathtt{ret}\ \rho_2(\mathbf{q'})\ \{\mathbf{r_2}\}, \cdot)]) \in \mathcal{O} \}
\end{aligned}
$$

$$
\text{ret-addr}_j(W, \mathbf{r}) = W.\mathbf{R_j}(\mathbf{r}) \quad \text{ret-addr}_j(W, \mathbf{i}) = W.\mathbf{S_j}(\mathbf{i}) \quad \text{ret-reg}_j(W, \mathbf{r}) = \mathbf{r'} \ \text{ if } W.\boldsymbol{\chi}_\mathbf{j}(\mathbf{r}) = \mathbf{box}\,\forall[].\{\mathbf{r'} : \boldsymbol{\tau}; \boldsymbol{\sigma'}\}^{\mathbf{q}}
$$

$$
\begin{aligned}
\boldsymbol{\Psi}; \boldsymbol{\Delta}; \boldsymbol{\Gamma}; \boldsymbol{\chi}; \boldsymbol{\sigma}; \mathbf{q} \vdash e_1 \approx e_2 : \tau; \boldsymbol{\sigma'} \ &\overset{\text{def}}{=}\ \forall W, \gamma, \rho.\ W \in \mathcal{H}[\![\boldsymbol{\Psi}]\!] \ \wedge\ \rho \in \mathcal{D}[\![\boldsymbol{\Delta}]\!] \ \wedge (W, \gamma) \in \mathcal{G}[\![\boldsymbol{\Gamma}]\!]\rho \ \wedge \text{curr-R}(W) \Subset \mathcal{R}[\![\boldsymbol{\chi}]\!]\rho \ \wedge \\
&\qquad \text{curr-S}(W) \Subset \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho \implies (W, \rho_1(\gamma_1(e_1)), \rho_2(\gamma_2((e_2))) \in \mathcal{E}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma'}]\!]\rho
\end{aligned}
$$

Figure 14. $\mathbb{FT}$ Logical Relation: Component and Continuation Relations and Equivalence of Open Terms

$$
\begin{aligned}
&\mathcal{HV}[\![\forall[\boldsymbol{\Delta}].\{\boldsymbol{\chi}; \boldsymbol{\sigma}\}^{\mathbf{q}}]\!]\rho = \\
&\ \{(W, \mathbf{code}[\boldsymbol{\Delta}]\{\rho_1(\boldsymbol{\chi}); \rho_1(\boldsymbol{\sigma})\}^{\rho_1(\mathbf{q})}.\mathbf{I_1}, \\
&\qquad \mathbf{code}[\boldsymbol{\Delta}]\{\rho_2(\boldsymbol{\chi}); \rho_2(\boldsymbol{\sigma})\}^{\rho_2(\mathbf{q})}.\mathbf{I_2}) \mid \\
&\qquad \forall W' \sqsupseteq W.\ \forall \rho^* \in \mathcal{D}[\![\boldsymbol{\Delta}]\!].\ \forall \boldsymbol{\tau}, \boldsymbol{\sigma'}. \\
&\qquad\quad \text{let } \rho' = \rho \cup \rho^* \text{ in } \boldsymbol{\tau}; \boldsymbol{\sigma'} =_{\rho'} \text{ret-type}(\mathbf{q}, \boldsymbol{\chi}, \boldsymbol{\sigma}) \ \wedge \\
&\qquad\quad \text{curr-R}(W') \Subset \mathcal{R}[\![\boldsymbol{\chi}]\!]\rho' \ \wedge\ \text{curr-S}(W') \Subset \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho' \\
&\qquad\quad \implies (W', (\rho_1^*(\mathbf{I_1}), \cdot), (\rho_2^*(\mathbf{I_2}), \cdot)) \in \mathcal{E}[\![\mathbf{q} \vdash \boldsymbol{\tau}; \boldsymbol{\sigma'}]\!]\rho' \}
\end{aligned}
$$

$$
\begin{aligned}
&\boldsymbol{\tau}; \boldsymbol{\sigma'} =_\rho \text{ret-type}(\mathbf{q}, \boldsymbol{\chi}, \boldsymbol{\sigma}) \ \overset{\text{def}}{=} \\
&\quad \rho_i(\boldsymbol{\tau}); \rho_i(\boldsymbol{\sigma'}) = \text{ret-type}(\rho_i(\mathbf{q}), \rho_i(\boldsymbol{\chi}), \rho_i(\boldsymbol{\sigma})), \ \text{ for } i \in 1, 2
\end{aligned}
$$

Figure 15. $\mathbb{FT}$ Logical Relation: Code Block

$\boldsymbol{\chi}$ and stack typing $\boldsymbol{\sigma}$ respectively. Related register files map registers to related values and related stacks are made up of related values. Stacks are related at the stack type $\boldsymbol{\zeta}$ if they are related by relational substitution ρ'.

Once we have related inputs, the logical relation should specify that applying the arguments produces related output expressions. Since the arguments are present in the registers and on the stack, we simply state that the instruction sequences $\mathbf{I_1}$ and $\mathbf{I_2}$, with empty heap fragments, are related components in the $\mathcal{E}$ relation under those conditions. In this, we rely critically on the return marker $\mathbf{q}$ to determine the return type $\boldsymbol{\tau}$ and resulting stack $\boldsymbol{\sigma'}$.

The logical relation $\mathcal{E}$ for components has three formal parameters: $\mathbf{q}$, $\boldsymbol{\tau}$, and $\boldsymbol{\sigma}$. The return marker $\mathbf{q}$ says where the expression is returning to as described in §3. The return type $\boldsymbol{\tau}$ is the type of value that is passed to the return continuation in $\mathbf{q}$, which is necessary in order to reason about equivalences, because if expressions don't even produce the same type of value they can't possibly be equivalent. This type comes from the ret-type metafunction whose definition is in Figure 2. The output stack type $\boldsymbol{\sigma}$ is also, in a sense, part of the return value and it is similarly derived from the return marker by the metafunctions.

The component relation $\mathcal{E}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma}]\!]\rho$ and relation for evaluation contexts $\mathcal{K}[\![\mathbf{q} \vdash \tau; \boldsymbol{\sigma}]\!]\rho$ are tightly connected, as is standard for logical relations based on biorthogonality. In typical biorthogonal presentations, the definitions would be:

$$
\begin{aligned}
\mathcal{K}[\![\tau]\!] &= \{(W, E_1, E_2) \mid \forall W'. W' \sqsupseteq W \wedge (W', v_1, v_2) \in \mathcal{V}[\![\tau]\!] \\
&\qquad\qquad \implies (W', E_1[v_1], E_2[v_2]) \in \mathcal{O}\} \\
\mathcal{E}[\![\tau]\!] &= \{(W, e_1, e_2) \mid \forall E_1 E_2. (W, E_1, E_2) \in \mathcal{K}[\![\tau]\!] \\
&\qquad\qquad \implies (W, E_1[e_1], E_2[e_2]) \in \mathcal{O}\}
\end{aligned}
$$

The above states that continuations E_1 and E_2 accepting type τ related at world W must be such that, given any future world W' and τ values, plugging in the values results in related observations. In turn, expressions e_1 and e_2 of type τ related at world W must be such that, given related continuations E_1 and E_2, $E_1[e_1]$ and $E_2[e_2]$ are observationally equivalent. Note how the reduction of e_1 and e_2 to values is central, since the definition of E_1 and E_2 tells you only that given related values they produce related observations. This reduction is normally captured in "monadic bind" lemmas.

Our definitions, in Figure 14, are more involved but follow a similar pattern. Our relation $\mathcal{E}$ only differs from the standard one in that the type of a component involves a return marker $\mathbf{q}$ and output stack type $\boldsymbol{\sigma}$.

The continuation relation $\mathcal{K}$ has three cases for different return marker $\mathbf{q}$. The case for $\mathbf{out}$, which corresponds to our functional terms, is nearly identical to the idealized case shown above. It differs only in requiring $\text{curr-S}(W') \in \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho$, which means that at the point we plug in the values $\mathsf{v_1}$ and $\mathsf{v_2}$ the stacks must be related at type $\boldsymbol{\sigma}$.

The $\mathcal{K}$ relation for $\mathbf{end\{\tau;\sigma\}}$ is similar, but since this is $\mathbb{T}$ code, return values are stored in registers $\mathtt{r_i}$ and the "value" being plugged in is the $\mathtt{halt}$ instruction.

The third case, when the return marker is a register $\mathtt{r}$ or a stack position $\mathbf{i}$, is more involved, though the overall meaning is still the same as the other cases: in the future, we will have a value to pass and will plug it into the hole to get related observations. First, we note that though at points during computation the return marker can be a stack index $\mathbf{i}$, when we actually return to the continuation the return marker must be stored in a register $\mathbf{q'}$. We require, however, that the code block being pointed to by $\mathbf{q}$ be the same as that pointed to by $\mathbf{q'}$. Next, we find the registers $\mathbf{r_1}, \mathbf{r_2}$ where the return values will be passed, and ensure that these contain related values. Finally, we check that the stacks are related at the right type with $\text{curr-S}(W') \in \mathcal{S}[\![\boldsymbol{\sigma}]\!]\rho$, before saying that plugging in the returns must yield related observations.

Having described how closed terms are related, we lift this to open terms with $\approx$, shown at the bottom of Figure 14. We choose appropriate closing type and term substitutions, where $\mathcal{G}[\![\boldsymbol{\Gamma}]\!]\rho$ is a relational substitution mapping $\mathbb{F}$ variables to related $\mathbb{F}$ values, and then state the equivalence after closing with these substitutions.

We have proven that the logical relation is sound and complete with respect to $\mathbb{FT}$ contextual equivalence (see technical appendix [21]).

Theorem 5.1 (Fundamental Property)
If $\boldsymbol{\Psi}; \boldsymbol{\Delta}; \boldsymbol{\Gamma}; \chi; \boldsymbol{\sigma}; \mathbf{q} \vdash e : \tau; \boldsymbol{\sigma'}$ then
$\boldsymbol{\Psi}; \boldsymbol{\Delta}; \boldsymbol{\Gamma}; \chi; \boldsymbol{\sigma}; \mathbf{q} \vdash e \approx e : \tau; \boldsymbol{\sigma'}$.

As usual, we prove compatibility lemmas corresponding to typing rules, after which the fundamental property follows as a corollary. While none of the compatibility lemmas for $\mathbb{T}$ instructions are trivial, the one for $\mathtt{call}$ is the most involved.

$$
\begin{aligned}
\mathsf{f_1} \quad &= \boldsymbol{\lambda}(\mathsf{x} : \mathsf{int}).^{(\mathsf{int}) \to \mathsf{int}}\mathcal{FT}(\mathtt{protect}\ \cdot, \zeta; \mathtt{mv}\,\mathtt{r1}, \ell;\\
&\qquad\qquad \mathtt{halt}\ (\mathsf{int}) \to \mathsf{int}^{\mathcal{T}}, \zeta\ \{\mathtt{r1}\},\\
&\qquad\qquad \mathbf{H_1})\ \mathsf{x}\\
\mathbf{H_1}(\ell) \ &= \mathbf{code}[\zeta, \epsilon]\{\mathtt{ra}\!:\!\forall[].\{\mathtt{r1}\!:\!\mathsf{int}^{\mathcal{T}}; \zeta\}^{\epsilon}; \mathsf{int}^{\mathcal{T}} :: \zeta\}^{\mathtt{ra}}.\\
&\quad \mathtt{sld}\,\mathtt{r1}, \mathbf{0}; \mathtt{add}\,\mathtt{r1}, \mathtt{r1}, \mathbf{1}; \mathtt{add}\,\mathtt{r1}, \mathtt{r1}, \mathbf{1};\\
&\quad \mathtt{sfree}\,\mathbf{1}; \mathtt{ret}\,\mathtt{ra}\,\{\mathtt{r1}\}\\
\mathsf{f_2} \quad &= \boldsymbol{\lambda}(\mathsf{x} : \mathsf{int}).^{(\mathsf{int}) \to \mathsf{int}}\mathcal{FT}\ (\mathtt{protect}\ \cdot, \zeta; \mathtt{mv}\,\mathtt{r1}, \ell;\\
&\qquad\qquad \mathtt{halt}\,\mathsf{int}^{\mathcal{T}}, \zeta\ \{\mathtt{r1}\}, \mathbf{H_2})\ \mathsf{x}\\
\mathbf{H_2}(\ell) \ &= \mathbf{code}[\zeta, \epsilon]\{\mathtt{ra}\!:\!\forall[].\{\mathtt{r1}\!:\!\mathsf{int}^{\mathcal{T}}; \zeta\}^{\epsilon}; \mathsf{int}^{\mathcal{T}} :: \zeta\}^{\mathtt{ra}}.\\
&\quad \mathtt{sld}\,\mathtt{r1}, \mathbf{0}; \mathtt{add}\,\mathtt{r1}, \mathtt{r1}, \mathbf{1}; \mathtt{sst}\,\mathbf{0}, \mathtt{r1}; \mathtt{jmp}\,\ell'[\zeta][\epsilon]\\
\mathbf{H_2}(\ell') &= \mathbf{code}[\zeta, \epsilon]\{\mathtt{ra}\!:\!\forall[].\{\mathtt{r1}\!:\!\mathsf{int}^{\mathcal{T}}; \zeta\}^{\epsilon}; \mathsf{int}^{\mathcal{T}} :: \zeta\}^{\mathtt{ra}}.\\
&\quad \mathtt{sld}\,\mathtt{r1}, \mathbf{0}; \mathtt{add}\,\mathtt{r1}, \mathtt{r1}, \mathbf{1}; \mathtt{sfree}\,\mathbf{1}; \mathtt{ret}\,\mathtt{ra}\,\{\mathtt{r1}\}
\end{aligned}
$$

Figure 16. $\mathbb{FT}$ Example: Different Number of Basic Blocks

In particular, $\mathtt{call}$ must ensure that the code that it is jumping to eventually returns, even while the target component could make nested calls. This relies on the target component return marker ensuring that control will eventually pass to the original return continuation.

Theorem 5.2 (LR Sound & Complete wrt Ctx Equiv)
$\boldsymbol{\Psi}; \boldsymbol{\Delta}; \boldsymbol{\Gamma}; \chi; \boldsymbol{\sigma}; \mathbf{q} \vdash e_1 \approx e_2 : \tau; \boldsymbol{\sigma'}$ if and only if
$\boldsymbol{\Psi}; \boldsymbol{\Delta}; \boldsymbol{\Gamma}; \chi; \boldsymbol{\sigma}; \mathbf{q} \vdash e_1 \approx^{ctx} e_2 : \tau; \boldsymbol{\sigma'}$.

5.1 Example Equivalences

In Figure 16, we show two programs that differ in the number of basic blocks that they use to carry out the same computation: adding two to a number and returning it. This example demonstrates our ability to reason over differences in internal jumps, which critically depends on the return markers explained in §3. We are able to show these two examples equivalent at type $(\mathsf{int}) \to \mathsf{int}$ using the logical relation. The elided proofs are included in the technical appendix [21].

In Figure 17, we show another small example. We present two implementations of the factorial function. The $\mathsf{fact_F}$ is a standard recursive functional implementation using iso-recursive types. We apply the function template F to a folded version of itself and the argument x. In the body, we check if the x is $\mathsf{0}$, in which case we return $\mathsf{1}$, and otherwise we unfold the first argument, call in with $\mathsf{x} - \mathsf{1}$, and multiply the result by x. This clearly produces the result for $\mathsf{x} \geq \mathsf{0}$, and also clearly diverges for negative arguments.

The imperative factorial $\mathsf{fact_T}$ uses registers to compute the result. It has two basic blocks, ℓ_{fact} and ℓ_{loop}. The first, which is translated to $\mathbb{F}$ and called with argument x, loads the argument $\mathbf{n}$ (translated from n) into register $\mathtt{r3}$, stores $\mathbf{1}$ in the result register $\mathtt{r7}$, and then checks if $\mathtt{r3}$ is $\mathbf{0}$. If so, we clear the argument off the stack and return. Otherwise, we jump to ℓ_{loop}. This multiplies the result by $\mathtt{r3}$, subtracts one from $\mathtt{r3}$, and makes the same check if $\mathtt{r3}$ is zero. If so,

$$
\begin{aligned}
\mathsf{fact_F} &= \lambda(\mathsf{x}:\mathsf{int}).(\mathsf{F}\ (\mathsf{fold}_{\mu\alpha.(\alpha)\to\mathsf{int}}\ \mathsf{F}))\ \mathsf{x}\\
\mathsf{F} &= \lambda(\mathsf{f}:\mu\alpha.(\alpha)\to\mathsf{int}).\lambda(\mathsf{x}:\mathsf{int}).\\
&\qquad \mathsf{if0}\ \mathsf{x}\ 1\ (((\mathsf{unfold}\ \mathsf{f})\ \mathsf{f})\ (\mathsf{x}-1))*\mathsf{x}\\
\mathsf{fact_T} &= \lambda(\mathsf{x}:\mathsf{int}).{}^{(\mathsf{int})\to\mathsf{int}}\mathcal{FT}\ (\\
&\qquad\quad \mathtt{protect}\ \cdot,\zeta;\mathtt{mv}\ \mathtt{r1},\ell;\\
&\qquad\quad \mathtt{halt}\ \mathsf{int}^{\mathcal{T}},\zeta\ \{\mathtt{r1}\},\\
&\qquad\quad \mathbf{H_2})\ \mathsf{x}\\
\mathbf{H}(\ell_{\mathrm{fact}}) &= \mathbf{code}[\zeta,\epsilon]\{\mathtt{ra}:\forall[].\{\mathtt{r1}:\mathsf{int}^{\mathcal{T}};\zeta\}^{\epsilon};\\
&\qquad\qquad\quad \mathsf{int}^{\mathcal{T}}::\zeta\}^{\mathtt{ra}}.\\
&\qquad \mathtt{sld}\ \mathtt{r3},0;\mathtt{mv}\ \mathtt{r7},1;\mathtt{bnz}\ \mathtt{r3},\ell_{\mathrm{loop}}[\zeta][\epsilon];\\
&\qquad \mathtt{sfree}\ 1;\mathtt{ret}\ \mathtt{ra}\ \{\mathtt{r7}\}\\
\mathbf{H}(\ell_{\mathrm{loop}}) &= \mathbf{code}[\zeta,\epsilon]\{\mathtt{r3}:\mathbf{int},\mathtt{r7}:\mathbf{int},\\
&\qquad\qquad\quad \mathtt{ra}:\forall[].\{\mathtt{r1}:\mathsf{int}^{\mathcal{T}};\zeta\}^{\epsilon};\\
&\qquad\qquad\quad \mathsf{int}^{\mathcal{T}}::\zeta\}^{\mathtt{ra}}.\\
&\qquad \mathtt{mul}\ \mathtt{r7},\mathtt{r7},\mathtt{r3};\mathtt{sub}\ \mathtt{r3},\mathtt{r3},1;\\
&\qquad \mathtt{bnz}\ \mathtt{r3},\ell_{\mathrm{loop}}[\zeta][\epsilon];\mathtt{sfree}\ 1;\mathtt{ret}\ \mathtt{ra}\ \{\mathtt{r7}\}
\end{aligned}
$$

Figure 17. $\mathbb{FT}$ Example: Factorial Two Different Ways

we do the same cleanup and return, and otherwise we jump to the beginning of ℓ_{loop} again.

While these two programs produce the same result, they do it in very different ways. First, $\mathsf{fact_F}$ uses recursive types, whereas $\mathsf{fact_T}$ does not. More importantly, $\mathsf{fact_F}$ uses a functional stack-based evaluation, whereas $\mathsf{fact_T}$ mutates registers and performs direct jumps. However, the proof of equivalence only differs from the proof for the example in Figure 16 in that we have to consider two cases — one in which they both diverge (for negative input n), and one in which they both terminate with related values (for non-negative input n).

6. Discussion and Future Work

FunTAL for Developers We have presented a multi-language $\mathbb{FT}$ that *safely* embeds assembly in a functional language. Moreover, our logical relation can be used to establish correctness of embedded assembly components. Developers of high-assurance software can write a high-level component e to serve as a specification for the TAL implementation $\mathbf{e}$ and use our logical relation to prove them equivalent.

$\mathbb{FT}$ also enables powerful compositional reasoning about high-level components, even in the presence of embedded assembly code. In fact, we conjecture that if the programmer does not use stack-modifying lambdas and if the embedded TAL contains no *statically defined* mutable tuples, then $\mathbb{FT}$ ensures referential transparency for high-level terms. Intuitively, in the absence of these side-channels (stack-manipulation and mutable cells), there is no way for two embedded TAL components to communicate with one another. Thus, even if a high-level term e contains embedded assembly, evaluating e has no observable effects. If the programmer does use stack-modifying lambdas or statically defined mutable tuples, reasoning about high-level components remains similar to reasoning about components in ML.

JIT Formalization We plan to investigate modeling a JIT compiler using multi-language programs. The high-level source language would be untyped and the low-level language would be typed assembly (since type information is precisely what a JIT runtime discovers about portions of high-level code, triggering compilation). We would consider the space of JIT optimization to be the set of possible replacements of untyped components with sound low-level versions, with appropriate guards included to handle violation of typing assumptions. Note, of course, that the low-level versions may still have calls back into high-level untyped code. What the JIT is then doing at runtime is moving between those configurations, usually by learning enough type information to make the guards likely to pass.

We can prove a JIT compiler correct based on the transformations that it would do. Formally, for all moves between configurations that the JIT may perform, we must show:

$$\forall E, \mathsf{e_S}.\ \mathsf{e_S} \overset{E}{\rightsquigarrow} \mathbf{e_T} \implies E[\mathsf{e_S}] \approx E[\mathcal{FT}\ \mathbf{e_T}]$$

where $\overset{E}{\rightsquigarrow}$ represents context-aware JIT-compilation that allows the compiler to use information in the context E, which could include values in scope, in order to decide how to transform a component $\mathsf{e_S}$ into $\mathbf{e_T}$. The definition of the JIT is thus $\overset{E}{\rightsquigarrow}$, and we would prove equivalence of the resulting multi-language programs using a logical relation similar to the one shown in this paper.

Compositional Compiler Correctness As mentioned in §1, Perconti and Ahmed [22] proved correctness of a functional-language compiler that performs closure conversion and heap allocation. We can easily adapt our multi-language to verify correctness of a code-generation pass from their allocation target $\mathbb{A}$ to $\mathbb{T}$, changing $\mathbb{FT}$ to $\mathbb{AT}$. The semantics of $\mathbb{T}$ and $\mathbb{T}$-relevant proofs in the logical relation can be reused without change. Correctness of code generation would then be expressed as contextual equivalence ($\approx^{ctx}$) in $\mathbb{AT}$: if $\mathsf{e_A}:\tau_\mathsf{A}$ compiles to $\mathbf{e_T}$ then $\mathsf{e_A} \approx^{ctx}\ {}^{\tau_\mathsf{A}}\mathcal{FT}\ \mathbf{e_T}$.

Continuation-Passing $\mathbb{F}$ and Rust Instead of trying to bridge the gap between the direct-style $\mathbb{F}$ and the continuation-aware $\mathbb{T}$, we could have made $\mathbb{F}$ a continuation-passing-style language, effectively lowering its level of abstraction to simplify interoperability with assembly. But the resulting multi-language would be more difficult for source programmers to use, as it would require them to reason about CPS'd programs. This is essentially the approach taken by the RustBelt project [9]—i.e., working with a Rust in continuation-passing style with embedded unsafe C.[2] The project seeks to establish soundness of Rust and its standard library, where the latter essentially contains unsafe embedded C. In contrast to $\mathbb{T}$, RustBelt does not take a multi-language approach or aim to handle inline assembly. Rather, it uses a sophisticated

[2] Personal communication with Derek Dreyer and Ralf Jung.

program logic for mutable state to reason about unsafe C code. It would be interesting to investigate a multi-language system with direct-style Rust interoperating with unsafe C and assembly along the lines of our work.

Choices in Multi-Language Design There are many potential choices when designing a multi-language system. For instance, we chose to expose low-level abstractions to high-level code by adding stack-modifying lambdas to $\mathbb{F}\mathbb{T}$ enabling more interactions between $\mathbb{F}$ and $\mathbb{T}$ code by invalidating equivalences that might otherwise have been used to justify correctness of compiler optimizations. We could also add foreign pointers to $\mathbb{F}\mathbb{T}$, which would allow references to mutable $\mathbb{T}$ tuples to flow into $\mathbb{F}$ as opaque values of lump type (as in Matthews-Findler [16]), allowing them to be passed but only used in $\mathbb{T}$. Foreign pointers would have the form ${}^{\mathsf{L}\langle\overline{\tau}\rangle}\mathcal{FT}\,\boldsymbol{\ell}$ (where $\boldsymbol{\ell} : \mathbf{ref}\ \langle\overline{\boldsymbol{\tau}}\rangle^{\mathcal{T}}$). While we can currently provide limited mutation to $\mathbb{F}$ via $\mathbb{T}$ libraries, foreign pointers would make that more flexible albeit at the cost of complicating the multi-language.

7. Related Work

There has been a great deal of work on multi-language systems, typed assembly languages, logics for modular verification of assembly code, and logical relations in general. We focus our discussion on the most closely related work.

Our work builds on results about typed assembly [17] and in particular STAL, its stack-based variant [18]. In §3 we explain in detail the differences between our TAL and STAL. Note here though, that these differences stem from our goal to use type structure to define the notion of a TAL component. We share this goal with a number of previous type-system design and verification efforts for flavors of assembly-like languages. Glew and Morrisett [12] tackle the problem of safe linking for TAL program fragments and provide an extension of TAL's type system that guarantees that linking preserves type safety. Benton [7] introduces a typed Floyd-Hoare logic for a stack-based low-level language that treats program fragments and their linking in a modular fashion. Outside the distinct technical details of what a component is in our TAL, our work differs from these results in that our notion of a TAL component matches that of a function in a high-level functional language.

Our multi-language semantics builds of work by Matthews and Findler [16] who give multiple interoperability semantics between a dynamically and statically typed language. We also build on multi-languages used for compiler correctness [2, 20, 22] which embed the source (higher-level) and target (lower-level) languages of a compiler, though none of that work considers interoperability with a language as low-level as assembly.

A related strand of research explores type safety and foreign function interfaces (FFI). Furr and Foster [11] describe sound type inference for the OCaml/C and JNI FFIs. Tan et al. [24] use a mixture of dynamic and static checks to construct a type-safe variant of JNI. Larmuseau and Clarke [15] aim for fully abstract and type-safe interoperability between ML and a low-level language. However, their model low-level language is Scheme with reflection. Tan [23] describes a core model for JNI that mixes Java bytecode and assembly. As an application, they design a sound type system for their multi-language. Our work is distinct as it captures how assembly interacts safely with a functional language.

Appel *et al.* showed how to prove soundness of TALs [5] using (unary) step-indexed models [3, 6]. Our logical relation most closely resembles the multi-language relation of Perconti and Ahmed [22] though theirs, without assembly, is simpler. Most prior logical relations pertaining to assembly or SECD machines are cross-language relations that specify equivalence of high-level (source) code and low-level (target) code and are used to prove compiler correctness [8, 13, 19]. Hur and Dreyer use a cross-language Kripke logical relation between ML and assembly to verify a one-pass compiler [13]. Neis *et al.* set up a parametric inter-language simulation (PILS) relating a functional source S and a continuation-passing-style intermediate language I, and one relating I to a target assembly T [19]. None of these can reason about equivalence of (multi-block) assembly components as we do. Jaber and Tabareau [14] present a logical relation indexed by source-language types but inhabited by SECD terms, capturing high-level structure. Besides being able to reason about mixed programs, our $\mathbb{F}\mathbb{T}$ logical relation—indexed by multi-language types—is more expressive: it can be used to prove equivalence of assembly components of type $\boldsymbol{\tau}$ when $\boldsymbol{\tau} = \tau^{\mathcal{T}}$ for some τ (analogous to Jaber-Tabareau) as well as when $\boldsymbol{\tau}$ *is not* of translation type. All of these logical relations make use of biorthogonality, a natural choice for continuation-based languages.

Finally, Wang *et al.* [25] describe a multi-language system in which components written in a C-like language can link with a simple untyped assembly, where the latter must be proven to adhere to a specification in higher-order logic. In their system, equivalences must be specified using axiomatic higher-order logic specifications. This differs significantly from FunTAL where equivalences arise out of extensional operational behavior with no external specification needed. Further, all of their assembly must be proven to follow an XCAP program specification, making it a much heavier approach than our typed assembly language. Our approach is complementary, in that while their higher-order logic allows finer grained specifications, it incurs additional cost on the programmers, and indeed renders it potentially non-viable for ML and x86 programmers that we believe would still be able to use FunTAL.

Acknowledgments

This research was supported in part by the National Science Foundation (grants CCF-1422133, CCF-1453796, CCF-1618732, CCF-1421770, and CNS-1524052) and a Google Faculty Research Award.

References

[1] A. Ahmed. Verified Compilers for a Multi-Language World. In T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15–31, 2015.

[2] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 431–444, Sept. 2011.

[3] A. Ahmed, A. W. Appel, and R. Virga. An indexed model of impredicative polymorphism and mutable references. Available at `http://www.cs.princeton.edu/~appel/papers/impred.pdf`, Jan. 2003.

[4] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*, Jan. 2009.

[5] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. *ACM Transactions on Programming Languages and Systems*, 32(3):1–67, Mar. 2010.

[6] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Nov. 2004.

[7] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS), Tsukuba, Japan*, pages 364–380, 2005.

[8] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*, Sept. 2009.

[9] D. Dreyer. RustBelt: Logical foundations for the future of safe systems programming. `http://plv.mpi-sws.org/rustbelt/`, 2016. Accessed: 2016-11-15.

[10] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4&5):477–528, 2012.

[11] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, Illinois*, pages 62–72, June 2005.

[12] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 250–261, 1999.

[13] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, Jan. 2011.

[14] G. Jaber and N. Tabareau. The journey of biorthogonal logical relations to the realm of assembly code. Workshop on Low-Level Languages (LOLA), `http://web.emn.fr/x-info/ntabareau/fichiers/lola2011.pdf`, 2011. Accessed: 2016-11-15.

[15] A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM) , York, UK*, pages 215–230, 2015.

[16] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*, pages 3–10, Jan. 2007.

[17] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 85–97, Jan. 1998.

[18] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

[19] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, Aug. 2015.

[20] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*, Sept. 2016.

[21] D. Patterson, J. Perconti, C. Dimoulas, and A. Ahmed. FunTAL: Reasonably mixing a functional language with assembly (technical appendix). Available at `http://www.ccs.neu.edu/home/amal/papers/funtal-tr.pdf`, Mar. 2017.

[22] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming (ESOP)*, Apr. 2014.

[23] G. Tan. JNI Light: An operational model for the core JNI. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems (APLAS), Shanghai, China*, pages 114–130, 2010.

[24] G. Tan, A. W. Appel, S. Chakradhar, R. Srivaths, A. Raghunathan, and D. Wang. Safe java native interface. In *Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[25] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 2014.