TEAL: a Total Expressive Assembly Language

A report on work in progress

Anonymous Author(s)

Abstract

We present the design of a typed assembly language with a property we dub *total correctness*: well-typed programs are guaranteed to return their specified values.

The design combines a high-level specification language with a low level executable langauge of instructions and blocks, connecting the two together in a single assembly language by means of typing rules.

Our language is total (i.e. every program terminates), but expressive, with support for higher-order functions and control flow that goes beyond primitive recursion.

The design is currently at an early stage, but we plan to extend it to support more sophisticated type systems, as well as effects and other features needed in a practical system.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Semantics**; • **Theory of computation** → **Type theory**.

Keywords: virtual machines, low-level languages, security, types, type systems, termination

ACM Reference Format:
Anonymous Author(s). 2018. TEAL: a Total Expressive Assembly Language: A report on work in progress. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 7 pages.

1 Introduction

A number of situations call for low-level languages with strong guarantees. For example, the Linux kernel supports extensibility by means of injection of user code written in the eBPF bytecode language [Rice 2023]. To prevent security violations, injected bytecode is statically analysed to ensure that it is type-safe and terminating. The static analyser checks for the absence of back references in program structure, for finite bounds on loops, and for various other syntactic properties that ensure termination but rather severely restrict the expressiveness of the language.

In a quite different context, in dependently typed programming languages such as Agda [Bove et al. 2009], programs can represent proofs of logical propositions, and consistency of the logic also requires checking that programs are type-safe and terminating. The type systems are sophisticated, and the languages are consequently very expressive, but existing implementations discard types at an early stage of compilation, weakening the guarantees offered by compiled code.

This article presents a preliminary design for TEAL, a typed, stack-oriented assembly language based on the SECD machine, which aims to avoid these shortcomings, combining type safety, expressiveness, termination guarantees, and a property that we call *total correctness*. The total correctness property involves attaching a behavioural specification to a TEAL program, and guarantees that each program meets its specification: that is, if the program type checks, it is guaranteed to return the specified value.

TEAL is designed as a compilation target, not a user-facing language. However, the guarantees it offers do not depend on the correctness of compilation, but arise directly from the type system that ties together specifications with instruction sequences and blocks.

Structure of the article. The rest of this article is structured as follows:

- Section 2 introduces our design by means of an example, Ackermann's function. The example shows that the language is based on Gödel's System T, and that it is typed and expressive, supporting higher-order computation and functions that cannot be expressed with primitive recursion, while maintaining termination guarantees.
- Section 3 gives the syntax, configurations and reductions of the language.
- Section 4 presents the specification language that describes computational behaviour of the instructions.
- Section 5 discusses our total correctness property, along with the more standard properties of type safety and termination.
- Section 6 discusses related work and some future plans.

The work we describe is not yet complete: we have a language design, but have not yet specified a compilation procedure or implemented the language. However, we have developed an embedding of the language in the Agda proof assistant that allows us to establish the guarantees of Section 5 with a high degree of confidence.

2 An example: Ackermann's function

The Ackermann function is a simple example of a function that is not primitive recursive. It takes two arguments (m, n) and can be defined succinctly as an iterative function over

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

```
1   succ     ({}, x:Nat ↦ suc x : Nat) : ...
2   one      ({}, _:Unit ↦ 1 : Nat) : ...
3   apply    ({f:Nat → Nat}, p:Nat ↦ f p : Nat) : ...
4   rep      ({f:Nat → Nat}, x:Nat ↦ _ : Nat) :
5            var x                          load x
6            suc                            increment x
7            rec one apply                  compute f^{x+1}(1)
8            ret
9   sucClo   ({}, _:Unit ↦ succ{} : Nat → Nat) :
10           clo 0 succ                     create closure succ{}
11           ret
12  repClo   ({}, A:Nat → Nat ↦ rep{A} : Nat → Nat) :
13           var A                          obtain function to repeat
14           clo 1 rep                      create closure rep{A}
15           ret
16  acker    ({}, x:Nat × Nat ↦ _ : Nat) :
17           var x                          x = (m, n)
18           fst                            obtain m
19           rec sucClo repClo              compute closure A_m
20           var x
21           snd                            obtain n
22           app                            apply closure A_m to n
23           ret
24  main     (· ⊢ acker{} (3,4) : Nat) :
25           clo 0 acker                    create closure acker{}
26           lit 3                          push 3
27           lit 4                          push 4
28           pair                           form pair (3,4)
29           app                            compute acker(3,4)
30           ret
```

Figure 1. Ackermann function

m:

$$A_0(n) = n + 1$$
$$A_{m+1}(n) = A_m^{n+1}(1)$$

where $f^n(x)$ denotes the n-fold application of a function f to an argument x:

$$f^0(x) = x$$
$$f^{n+1}(x) = f(f^n(x))$$

Alternatively, these iterative functions may be equivalently written using an explicit *recursor*, that is, a construct $\mathbf{Rec}(z, (x, p).s, m)$ that interprets a natural number m using terms z and s, yielding z when m is 0 and s when m is $1 + x$. The sub-term $(x, p).s$ makes the variables x and p available within the expression s, binding x to the predecessor of m and p to the result of applying the recursor to x. The n-fold application is expressed as follows

$$f^n(x) \equiv \mathbf{Rec}(x, (\_, p).\ p\ x, n)$$

and the Ackermann function is defined as follows

$$A_m \equiv \mathbf{Rec}((\lambda n.n + 1), (\_, A).(\lambda n.\ A^{n+1}(1)), m)$$
$$\mathsf{ack}\ (m, n) \equiv A_m(n)$$

where the recursor returns a function of type Nat → Nat.

Figure 1 shows the assembly code corresponding to the definition of ack. [1] Each label corresponds to a function or a zero/successor case of recursor in ack, and the signature attached to each label specifies the behaviour of the instructions that follow. For example, succ (Line 1) corresponds to the successor function $\lambda n.n + 1$, rep (Line 4) corresponds to $(\lambda n.\ A^{n+1}(1))$, and the two recursor cases in the definition of A_m are represented by sucClo and repClo, each of which returns a function closure. The signature for acker (Line 16, omitted in the figure) is

$$(\mathbf{Rec}(\mathsf{suc}\{\}, (\_, A).\mathsf{rep}\{A\}, \mathbf{fst}\ x))\ (\mathbf{snd}\ x)$$

i.e. the unfolded definition of $A_m(n)$ with functions replaced by labels and explicit projections out of the pair.

3 Assembly language and stack machine

Our assembly language TEAL is an instruction set for a SECD virtual machine [Landin 1964], where SECD stands for four of the machine components: stack (st), environment (env), code (ls), and dump, which in modern terms, is a stack of function call frames (fr).

Instructions	I	::=	`var x` | `clo` $n\ \mathfrak{L}$ | `app`
			| `pair` | `fst` | `snd`
			| `lit` n | `suc` | `rec` $\mathfrak{L}_z\ \mathfrak{L}_s$
Sequences	Is	::=	`ret` | I; Is
Code block	$\mathfrak{B}$	::=	$\cdot$ | $\mathfrak{B}$; $\mathfrak{L}$({Γ}, x:A ↦ M : B) : Is
Program	P	::=	$\mathfrak{B}$; main(Γ ⊢ M : A) : Is

Figure 2. Syntax of TEAL

3.1 Syntax and machine configuration

The syntax of TEAL is shown in Figure 2. The instructions can be categorized in the following way:

- Variables: `var` (access)
- Closures: `clo` (formation), `app` (application)
- Pairs: `pair` (formation), `fst`, `snd` (projections)
- Natural numbers: `lit` (literal constant), `suc` (increment), `rec` (bounded recursion)

Instruction `var x` copies the x-th element from the environment to the stack. Instruction `clo` $n\ \mathfrak{L}$ takes the top n items from the stack to form a closure with code label $\mathfrak{L}$. The rec instruction takes two labels for a bounded recursion on the natural number on top of the stack, where $\mathfrak{L}_z$ is the code

[1] For simplicity, the signatures do not strictly follow the typing rules.

label for the zero case and $\mathfrak{L}_s$ is the label for the successor case. Our framework supports other useful instructions like swap and stack access, but we omit them here for simplicity.

The instructions can be chained into a sequence that ends with `ret` (the return instruction). A code block $\mathfrak{B}$ is a list of labelled instruction sequences with their specifications (which we will cover in Section 4.1), and a program is a code block with a main sequence.

Values	v	::=	$n \mid \langle\rangle \mid \mathfrak{L}\{\overline{v}\} \mid \langle v, v'\rangle$
Runtime environment	env	::=	$\cdot \mid \text{env} :: v$
Runtime stack	st	::=	$\cdot \mid \text{st} :: v$
Call frames	fr	::=	$\cdot \mid \text{fr} :: \langle \text{ls}, \text{env}, \text{st}\rangle$
Machine configuration	C	::=	$\langle \text{ls}, \text{env}, \text{st}, \text{fr}\rangle_{\mathfrak{B}}$

Figure 3. Machine configuration

Figure 3 shows the definition of the SECD abstract machine. A machine configuration $\langle \text{ls}, \text{env}, \text{st}, \text{fr}\rangle_{\mathfrak{B}}$ consists of the four SECD components and a code block. The stack and the environment are lists of runtime values, which include natural numbers, the unit value (dummy), pairs, and closures (a code label paired with a list of values).

3.2 Operational semantics

The small-step operational semantics of the instructions are summarized in Table 1. For each instruction, if the machine is in a state of the correct form and satisfies the condition (described in the first row of each entry), then it steps into the next state (the second row in each entry). For example, if the top instruction is `var x` and the x-th variable is found on the environment (i.e. $\text{env}(x) = v$), then the associated value v is pushed on the stack and the machine moves on to the next instruction.

Application and return. To execute an application, the stack should contain a closure and an argument on the top, where the closure's label points to an instruction sequence ls' in the code block $\mathfrak{B}$. Then, the machine clears the stack, loads the new instruction sequence ls' and the new environment formed by the values in the closure and the argument. The current instructions, environment, and stack are saved on a new call frame.

To return from a call, the stack must have a return value on the top, and the machine restores the saved instructions, environment, and stack from the call frames, then pushes the return value on the restored stack.

Recursion. To execute recursion, the top stack item must be a natural number. If the number is 0, the machine loads the instructions pointed by the base case label $\mathfrak{L}_z$. To simplify the calling mechanism, TEAL does not support loading a piece of code directly, but we can encode this feature as feeding a function with a dummy input (the unit value) — the machine creates a closure with $\mathfrak{L}_z$ and the current environment, pushes the dummy value $\langle\rangle$ on the stack, and immediately applies them.

When the top stack item equals $n = m + 1$, the machine makes a closure for $\mathfrak{L}_s$ with the current environment and m, adds an `app` instruction to the instruction sequence, and continues recursing over m (rule RecS). Every round of recursion creates a closure for the successor case on the stack and leaves an application instruction in the sequence. Eventually, after reaching the base case, the successor case will then be executed n times through applications.

Types	A, B, C	::=	$\mathbf{Unit} \mid \mathbf{Nat} \mid A \to B \mid A \times B$
Expressions	M, N	::=	$x \mid L\ M \mid \mathfrak{L}\{\overline{M}\}$
			$\mid \langle M, N\rangle \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M$
			$\mid \langle\rangle \mid \mathbf{zero} \mid \mathbf{suc}\ M$
			$\mid \mathbf{Rec}(M, (x, p).M', N)$
Type contexts	Γ	::=	$\cdot \mid \Gamma, x{:}A$
Label contexts	$\mathfrak{D}$	::=	$\cdot \mid \mathfrak{D}, \mathfrak{L}(\{\Gamma\}, x{:}A \mapsto M : B)$

Figure 4. Syntax of the specification language

Notations. We write $C \longrightarrow C'$ if configuration C steps to C' in one step, and $C \longrightarrow^* C'$ if C steps to C' in zero or more steps. A machine terminates with value v if the instruction is `ret`, the top stack value is v, and there is no call frame left.

4 Typed assembly language

In the example of the Ackermann function (Figure 1), each code block has a type signature that specifies its intended behaviour: repClo should return a closure, main should compute acker{} (3, 4), etc.

In this section, we define the language of specifications, a calculus that fully captures the computational aspects of the assembly. We then give typing rules that simulate the execution of the assembly code to check if the instructions indeed meet the specification.

4.1 The specification language

The specification language of TEAL is a variation of Gödel's System T based on [Huang and Yallop 2023]. The syntax of the language (Figure 4) includes the conventional variables, application, construction and projection for pairs, and natural numbers with their recursor (as introduced in Section 2). Our main departure is in the way that functions are expressed.

Function labels. In conventional lambda calculus, we can create functions with a lambda abstraction $\lambda x.M$ and evaluate function applications with the substitution $(\lambda x.M)\ N \triangleright M[N/x]$, also known as the β-reduction. Since the assembly language cannot arbitrarily introduce new functions, we use

Rule	ls	env	st	fr	Condition
Var	var x ; ls	env	st	fr	env(x) = v
⟶	ls	env	st :: v	fr	
Clo	clo n $\mathfrak{L}$; ls	env	st :: $\overline{v}$	fr	$\lvert\overline{v}\rvert = n$
⟶	ls	env	st :: $\mathfrak{L}\{\overline{v}\}$	fr	
App	app ; ls	env	st :: $\mathfrak{L}\{env'\}$ :: v	fr	$\mathfrak{B}(\mathfrak{L}) = ls'$
⟶	ls′	env′ :: v	·	fr :: ⟨ls, env, st⟩	
Ret	ret	env	st :: v	fr :: ⟨ls′, env′, st′⟩	
⟶	ls′	env′ :: v	st′	fr	
Pair	pair ; ls	env	st :: v :: v′	fr	
⟶	ls	env	st :: ⟨v, v′⟩	fr	
Fst	fst ; ls	env	st :: ⟨v, v′⟩	fr	
⟶	ls	env	st :: v	fr	
Snd	fst ; ls	env	st :: ⟨v, v′⟩	fr	
⟶	ls	env	st :: v′	fr	
RecZ	rec $\mathfrak{L}_z$ $\mathfrak{L}_s$; ls	env	st :: n	fr	$n = 0$
⟶	app ; ls	env	st :: $\mathfrak{L}_z\{env\}$ :: ⟨⟩	fr	
RecS	rec $\mathfrak{L}_z$ $\mathfrak{L}_s$; ls	env	st :: n	fr	$n = m + 1$
⟶	rec $\mathfrak{L}_z$ $\mathfrak{L}_s$; app ; ls	env	st :: $\mathfrak{L}_s\{env :: m\}$ :: m	fr	

Table 1. Operational semantics

a *defunctionalized* calculus that forbids lambda abstraction. The only way to create functions is to form a closure with a list of expressions and a code label $\mathfrak{L}$. The labels come from a globally defined label context $\mathfrak{D}$ that associates each label with its type information and an expression that acts like the function body.

A label-context entry $\mathfrak{L}(\{\Delta\}, x{:}A \mapsto L : B)$ is interpreted as follows: $\mathfrak{L}$ points to a piece of code with free variables of types Δ (we view a context as a list of types), takes an input x of type A, and the associated computation on x is N of type B. A closure $\mathfrak{L}\{\overline{M}\}$ is interpreted as follows: the free variables of $\mathfrak{L}$ are instantiated to $\overline{M}$, and the application $\mathfrak{L}\{\overline{M}\}$ N evaluates to $L[\overline{M}/\Delta, N/x]$ — substituting the instantiated free variables and the argument into the function body.

Note that the syntax of a label-context entry is identical to an assembly code block's signature and a label context is regarded as a collection of the specifications of an assembly program.

Typing judgement. The types in the calculus includes natural numbers, the unit type, products, and functions. We use a inference-based judgement to define well-typed terms for the specification calculus (Figure 6). The judgement takes the form $\mathfrak{D}; \Gamma \vdash M : A$, which means that expression M is well-typed with type A under label context $\mathfrak{D}$ and type context Γ (a list of variables and their associated types).

Figure 5 shows the typing rules: if the judgements or conditions above the line are derivable or satisfied, then the rule allows us to derive the judgement at the bottom. For example, if we know that M has the product type $A \times B$, then the rule rule TY-FST allows us to derive that the first projection of M has type A.

Rule TY-LABEL states that a label formation is well-typed if the label's specification exists in $\mathfrak{D}$ and the types of the list of expressions supplied match with the specification.

A recursor is well-typed if the term N to recurse on is a natural number, the zero case M is a well-typed expression (of some result type C), the successor case M′ is typed with C in the context extended by the predecessor x:Nat and r:C, the result of result of applying the recursor to x (TY-REC).

$\boxed{\vdash \mathfrak{D}}$ *(Well-formedness)*

$$\textsc{wf-Empty}\ \frac{}{\vdash \cdot} \qquad \textsc{wf-Label}\ \frac{\mathfrak{D}; \Delta, x{:}A \vdash M : B}{\vdash \mathfrak{D}, \mathfrak{L}(\{\Delta\}, x{:}A \mapsto N : B)}$$

The judgement for well-typed label context is $\vdash \mathfrak{D}$, which is required in all base cases like rule TY-VAR and rule TY-UNIT to ensure that type judgements are always considered in well-typed label contexts.

4.2 Typed stack machine

The specification calculus captures every kind of computation of the assembly. It gives us a way to describe the behaviour of an instruction sequence abstractly. For example, we know that var x copies the *unknown* value x on top of the stack, and app applies the top two stack items together.

$\boxed{\mathfrak{D}; \Gamma \vdash M : A}$ *(Typing)*

$$\textsc{ty-Var}\ \frac{x : A \in \Gamma \quad \vdash \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash x : A} \qquad \textsc{ty-Unit}\ \frac{\vdash \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \langle\rangle : \mathbf{Unit}}$$

$$\textsc{ty-Apply}\ \frac{\mathfrak{D}; \Gamma \vdash M : A \to B \quad \mathfrak{D}; \Gamma \vdash N : A}{\mathfrak{D}; \Gamma \vdash M\ N : B} \qquad \textsc{ty-Label}\ \frac{\mathfrak{D}; \Gamma \vdash \overline{M} : \Delta \quad \mathfrak{L}(\{\Delta\}, x{:}A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \mathfrak{L}\{\overline{M}\} : A \to B}$$

$$\textsc{ty-Fst}\ \frac{\mathfrak{D}; \Gamma \vdash M : A \times B}{\mathfrak{D}; \Gamma \vdash \mathbf{fst}\ M : A} \qquad \textsc{ty-Snd}\ \frac{\mathfrak{D}; \Gamma \vdash M : A \times B}{\mathfrak{D}; \Gamma \vdash \mathbf{snd}\ M : B}$$

$$\textsc{ty-Rec}\ \frac{\mathfrak{D}; \Gamma \vdash N : \mathbf{Nat} \quad \mathfrak{D}; \Gamma \vdash M : C \quad \mathfrak{D}; \Gamma, x{:}\mathbf{Nat}, p{:}C \vdash M' : C}{\mathfrak{D}; \Gamma \vdash \mathbf{Rec}(M, (x, p).M', N) : C}$$

Figure 5. Typing rules for the specification

Typing rules for assembly. We capture computations using the judgement $\mathfrak{D}; \Gamma \vdash \mathsf{Is} : \sigma \to \sigma'$ (Figure 6). We view σ, a list of *open terms* in the specification language, as an abstract stack. The judgement reads: the instruction (sequence) transits abstract stack from σ to σ' under the presence of some code blocks with signatures $\mathfrak{D}$ and some runtime environment with values of types Γ.

The rules are then straightforward. The closure formation `clo` $n\ \mathfrak{L}$ is modelled by $\mathfrak{L}\{\overline{M}\}$ (ASM-TY-CLO), the application `app` by the application in the calculus (ASM-TY-APPLY), and recursion `rec` $\mathfrak{L}_z\ \mathfrak{L}_s$ by the recursor (ASM-TY-REC), etc.

Well-formed machines. Notice that the stack machine's runtime values coincide with the *closed terms* in the calculus. We can take a step further and extend the type judgements to runtime values, environments, stacks, and machine configurations. We write $\vdash C$ for the judgement of well-formed machine configurations, and leave the technical details in appendix A. Intuitively, $\vdash \langle \mathsf{Is}, \mathsf{env}, \mathsf{st}, \mathsf{fr}\rangle_{\mathfrak{B}}$ means that:

1. All signatures and instructions in code blocks $\mathfrak{B}$ are well-typed, with $\mathfrak{D}$ being the signature.
2. The current instruction sequence Is is well-typed and produces a return value, i.e. $\mathfrak{D}; \Gamma \vdash \mathsf{Is} : \sigma \to \sigma' :: M$.
3. The runtime environment env has types Γ.
4. The runtime stack st is a concrete implementation of σ with respect to env.
5. The current instruction's return value M is compatible with its caller on the top frame of fr.

$\boxed{\mathfrak{D}; \Gamma \vdash \mathsf{Is} : \sigma \to \sigma'}$ *(Typing)*

$$\textsc{asm-ty-Var}\ \frac{x : A \in \Gamma}{\mathfrak{D}; \Gamma \vdash \mathtt{var}\ x : \sigma \to \sigma :: x}$$

$$\textsc{asm-ty-Clo}\ \frac{\mathfrak{D}; \Gamma \vdash \overline{M} : \Delta \quad |\Delta| = n \quad \mathfrak{L}(\{\Delta\}, x{:}A \mapsto M : B) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \mathtt{clo}\ n\ \mathfrak{L} : \sigma :: \overline{M} \to \sigma :: \mathfrak{L}\{\overline{M}\}}$$

$$\textsc{asm-ty-Apply}\ \frac{\mathfrak{D}; \Gamma \vdash M : A \to B \quad \mathfrak{D}; \Gamma \vdash N : A}{\mathfrak{D}; \Gamma \vdash \mathtt{app} : \sigma :: M :: N \to \sigma :: M\ N}$$

$$\textsc{asm-ty-Rec}\ \frac{\mathfrak{D}; \Gamma \vdash N : \mathbf{Nat} \quad \mathfrak{L}_z(\{\Gamma\}, x{:}\mathbf{Unit} \mapsto M : C) \in \mathfrak{D} \quad \mathfrak{L}_s(\{\Gamma, x{:}\mathbf{Nat}\}, p{:}C \mapsto M' : C) \in \mathfrak{D}}{\mathfrak{D}; \Gamma \vdash \mathtt{rec}\ \mathfrak{L}_z\ \mathfrak{L}_s : \sigma :: N \to \sigma :: \mathbf{Rec}(M, (x, p).M', N)}$$

$$\textsc{asm-ty-Seq}\ \frac{\mathfrak{D}; \Gamma \vdash \mathsf{I} : \sigma \to \sigma' \quad \mathfrak{D}; \Gamma \vdash \mathsf{Is} : \sigma' \to \sigma''}{\mathfrak{D}; \Gamma \vdash \mathsf{I}\ ;\ \mathsf{Is} : \sigma \to \sigma''}$$

$$\textsc{asm-ty-Ret}\ \frac{}{\mathfrak{D}; \Gamma \vdash \mathtt{ret} : \sigma :: M \to \sigma :: M}$$

Figure 6. Typing rules for the assembly

5 Total correctness

In this section, we discusses our total correctness property and its relations with the standard properties of type safety and termination.

For a typed system, type-safety means that if a program is well-typed, then it will never get stuck at some erroneous state. It is typically proved by proving progress (that well-typed program either halts or takes one step) and preservation (that a well-typed program remains well-typed after taking one step). For our system, we consider progress and preservation for well-formed machines.

Theorem 5.1 (Progress). *If $\vdash C$, then either $C \longrightarrow C'$ or C terminates.*

Theorem 5.2 (Preservation). *If $\vdash C$ and $C \longrightarrow C'$, then $\vdash C'$.*

The two theorems are proved by induction over the well-formedness judgement $\vdash C$. With the progress and preservation combined, we can see that a well-typed program either halts or steps into another well-typed state, which will again

either halts or keeps executing. In other words, a well-formed machine will never get stuck.

Corollary 5.3 (Type safety). *If $\vdash C$, then there is no stuck configuration C' such that $C \longrightarrow^* C'$.*

Since we specified the exact computational behaviour of an instruction sequence, we naturally ask not just that "well-typed programs will not go wrong", but also that "well-typed programs will do right" — if an instruction sequence is specified with return value v by its typing annotation, then executing it is guaranteed to produce v on the top of the stack. We call this property *total correctness*.

With our definition of well-formed machines, if we have type safety, then we obtain *partial correctness* that guarantees a program to compute the specified value if it terminates. Indeed, if we have a well-typed program

$$\mathfrak{D}; \cdot \vdash \mathsf{Is} : \cdot \longrightarrow \sigma$$

and the execution of $\langle \mathsf{Is}, \cdot, \cdot, \cdot \rangle_{\mathfrak{B}}$ terminates, then the final state must be in the form of $\langle \mathtt{ret}, \cdot, \mathtt{st}', \cdot \rangle_{\mathfrak{B}}$. By type-safety, the final state is well-formed, which means that $\mathtt{st}'$ is a concrete implementation of $(\mathtt{st} :: \mathsf{v})$ w.r.t. the empty context, which implies that the top stack item equals to v. So, if all well-typed programs terminate, then we obtain total correctness as a corollary from type safety and termination.

To show termination, we follow Benton and Hur [2009] in defining a *logical relation* (a type-indexed subset) which contains all stack-instruction pairs that terminate when they are executed with well-typed environment and call frames, and show that if $\mathfrak{D}; \cdot \vdash \mathsf{Is} : \cdot \longrightarrow \sigma :: \nu$, then $(\cdot, \mathsf{Is})$ is in that relation.

Theorem 5.4 (Termination). *If $\mathfrak{D}; \cdot \vdash \mathsf{Is} : \cdot \longrightarrow \sigma :: \nu$, then the machine terminates, i.e. $\langle \mathsf{Is}, \cdot, \cdot, \cdot \rangle_{\mathfrak{B}} \longrightarrow^* \langle \mathtt{ret}, \cdot, \mathtt{st}', \cdot \rangle_{\mathfrak{B}}$.*

Corollary 5.5 (Total correctness). *If $\mathfrak{D}; \cdot \vdash \mathsf{Is} : \cdot \longrightarrow \sigma :: \nu$, then $\langle \mathsf{Is}, \cdot, \cdot, \cdot \rangle_{\mathfrak{B}} \longrightarrow^* \langle \mathtt{ret}, \cdot, \mathtt{st} :: \nu, \cdot \rangle_{\mathfrak{B}}$.*

6 Discussion

Related work. Typed intermediate languages have been known to be useful in program analysis and optimization since Tarditi et al. [1996]. We closely follow the development of typed assembly languages (TAL) [Morrisett et al. 1998], first developed by Morrisett et al. [1998] as a compilation target to preserve the type information from ML-like type systems to the assembly level and focused on assigning types to the assembly. The type safety of TAL is established syntactically in Morrisett et al. [1998] and in many of its variations [Crary et al. 1999; Morrisett et al. 2002], as well as semantically in Ahmed et al. [2010]. We are not aware of any study on the termination of TAL.

Our type system can be regarded as a relation between the high-level specification calculus and the low-level machine configuration. The type system of Shao et al. [2002] relates terms in a dependently typed calculus with TAL. Benton and Hur [2009] relates simply-typed lambda calculus with an untyped SECD machine and established correctness for their compilation scheme from the calculus to the assembly.

Compilation. The language in this article is designed as a compilation target, but we have not yet specified the compilation function. Compilation from Gödel's System T to TEAL consists of a defunctionalization stage [Reynolds 1972] that replaces lambda abstractions with labels, targeting the specification language of Section 4.1, followed by a type-preserving translation from the specification language to assembly code. In future work we plan to specify and implement compilation.

Conclusions and Future work. We have presented a typed total assembly language based on Gödel's System T. The principles underlying the design of our language naturally extend to a wider class of typed languages, and we plan to investigate their application to a variety of more expressive systems, including

- *polymorphic lambda calculus* [Reynolds 1974], which forms a basis for functional programming languages such as Haskell, OCaml and Scala
- the *calculus of constructions* [Coquand and Huet 1988] and other dependently typed systems, which underlie dependently typed programming languages such as Agda and Idris, and proof assistants such as Rocq and Lean
- the *computational meta-language* [Moggi 1991] and other effective calculi which support programs that perform effects, and
- systems such as *quantitative type theory* [Atkey 2018] that offer support for distinguishing logical terms from computations, enabling efficient run-time representations.

We envisage a family of typed expressive assembly languages that make it possible to preserve types through compilation for these and other calculi, offering stronger safety guarantees and new optimisation opportunities.

References

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 97–108. https://doi.org/10.1145/1596550.1596567

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in*

Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674), Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 25–35.

Yulong Huang and Jeremy Yallop. 2023. Defunctionalization with Dependent Types. *Proc. ACM Program. Lang.* 7, PLDI (2023), 516–538. https://doi.org/10.1145/3591241

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-based typed assembly language. *J. Funct. Program.* 12, 1 (2002), 3–88. https://doi.org/10.1017/S0956796801004178

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 85–97. https://doi.org/10.1145/268946.268954

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, John J. Donovan and Rosemary Shields (Eds.). ACM, 717–740. https://doi.org/10.1145/800194.805852

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard J. Robinet (Ed.). Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

Liz Rice. 2023. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media. ISBN 978-1098135126.

Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 2002. A type system for certified binaries. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 217–232. https://doi.org/10.1145/503272.503293

David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 181–192. https://doi.org/10.1145/231379.231414

A Configuration well-formedness

$\boxed{\mathfrak{D} \vdash \mathfrak{B}}$ *(Code block typing)*

$$\textsc{asm-blocks-Nil}\ \frac{}{\cdot \vdash \cdot}$$

$$\textsc{asm-blocks-Cons}\ \frac{\mathfrak{D} \vdash \mathfrak{B} \qquad \mathfrak{D}; \Gamma, x{:}A \vdash \mathsf{Is} : \cdot \longrightarrow \sigma :: M}{\mathfrak{D}, \mathfrak{L}(\{\Gamma\}, x{:}A \mapsto M : B) \vdash \mathfrak{B}; \mathfrak{L}(\{\Gamma\}, x{:}A \mapsto M : B) : \mathsf{Is}}$$

$\boxed{\mathfrak{D} \vdash \mathsf{env} : \Gamma}$ *(Runtime environment typing)*

$$\textsc{asm-env-Nil}\ \frac{}{\mathfrak{D} \vdash \cdot : \cdot} \qquad \textsc{asm-env-Cons}\ \frac{\mathfrak{D} \vdash \mathsf{env} : \Gamma \qquad \mathfrak{D}; \cdot \vdash v : A}{\mathfrak{D} \vdash \mathsf{env} :: v : \Gamma, x{:}A}$$

$\boxed{\mathfrak{D}; \mathsf{env} : \Gamma \vdash \mathsf{st} : \sigma}$ *(Runtime stack typing)*

$$\textsc{asm-st-Nil}\ \frac{}{\mathfrak{D}; \mathsf{env} : \Gamma \vdash \cdot : \cdot} \qquad \textsc{asm-st-Cons}\ \frac{\mathfrak{D}; \mathsf{env} : \Gamma \vdash \mathsf{st} : \sigma \qquad \mathfrak{D}; \Gamma \vdash M : A \qquad \mathfrak{D}; \cdot \vdash v : A \qquad \mathfrak{D} \vdash v \equiv M[\mathsf{env}]}{\mathfrak{D}; \mathsf{env} : \Gamma \vdash \mathsf{st} :: v : \sigma :: M}$$

$\boxed{\mathfrak{D}; \mathsf{env} : \Gamma \vdash \mathsf{fr} : M}$ *(Call frames typing)*

$$\textsc{asm-frame-Nil}\ \frac{\mathfrak{D}; \cdot \vdash v : A}{\mathfrak{D}; \cdot : \cdot \vdash \cdot : v} \qquad \textsc{asm-frame-Cons}\ \frac{\begin{array}{c}\mathfrak{D}; \mathsf{env} : \Gamma \vdash \mathsf{fr} : M \\ \mathfrak{D}; \Delta \vdash \mathsf{Is} : \sigma :: N \longrightarrow \sigma' :: N' \\ \mathfrak{D} \vdash \mathsf{env}' : \Delta \\ \mathfrak{D}; \mathsf{env}' : \Delta \vdash \mathsf{st} : \sigma \\ \mathfrak{D} \vdash N'[\mathsf{env}'] \equiv M[\mathsf{env}]\end{array}}{\mathfrak{D}; \mathsf{env}' : \Delta \vdash \mathsf{fr} :: \langle \mathsf{Is}, \mathsf{env}', \mathsf{st} \rangle : N}$$

Definition A.1 (Configuration well-formedness)**.** *A configuration* $C = \langle \mathsf{Is}, \mathsf{env}, \mathsf{st}, \mathsf{fr} \rangle_{\mathfrak{B}}$ *is well-formed, denoted as* $\vdash C$*, if the following judgements hold:*

1. $\mathfrak{D} \vdash \mathfrak{B}$*, where* $\mathfrak{D}$ *is the label context of* $\mathfrak{B}$*.*
2. $\mathfrak{D}; \Gamma \vdash \mathsf{Is} : \sigma \longrightarrow \sigma' :: M$*, for some* Γ*,* σ*,* σ'*, and* M*.*
3. $\mathfrak{D} \vdash \mathsf{env} : \Gamma$*.*
4. $\mathfrak{D}; \mathsf{env} : \Gamma \vdash \mathsf{st} : \sigma$*.*
5. $\mathfrak{D}; \mathsf{env}' : \Delta \vdash \mathsf{fr} : N$*, for some* env'*,* Δ*, and* N*.*
6. $\mathfrak{D} \vdash N[\mathsf{env}'] \equiv M[\mathsf{env}]$*.*