Verifying a Postfix Language

Dustin Van Tate Testa and Andrew Neth

August 1, 2023

1 Introduction

1.1 Motivation

The functional, postfix language this paper focuses on is currently being used to power some insignificant demos, but maybe in some distant future it will end up as the backend for a visual programming tool or some other application where guarantees of safety like the ones here would be very important.

1.2 The Language

This section describes some of the key features of the original language. It is included only to show that the subset we chose is a good representation of the original language and thus can be skipped.

1.2.1 Full Syntax

- **Primitive Literals:** The language supports number and string literals
- **Closures:** closures aka macros are enclosed within `(: ... )` and can have type annotations, for example `((I32 I32) (I32) rec: ... )`.
- **Tuples:** Tuples are enclosed within parenthesis. There is no reason to include separators like commas.
- **Identifiers:** there are two types of identifiers, escaped identifiers which start with `$` are generally used for assignment and unescaped identifiers which are used for to invoke the value to which the identifier refers.
- **Functions:** function overloading is the only supported form of branching in the language. The syntax for the `fun` operator is as follows.

  `((`$\ldots_1$`)(`$\ldots_2$` I32): Condition ) ((`$\ldots_1$`): Action ) $Identifier fun`.
- **Other Operators:** Although the majority of primitive operators are defined in the standard library, the compiler actually has a few builtins as well. Although these are very important to actually using the language, they aren't worthy of listing out in their entirety here. Some examples include: `require`, `use`, `import`, `export`, `asm`, `namespace`, `type`, `unpack`, `class`, `=`, `|`, `@`, `~`, `==`, `make`

Combining these concepts we can make the following program.

```
# Import basic math and logic
"stdlib/prelude.phs" require use

# Define a recursive factorial closure
((I32)(I32) rec:
    $n =
    (: 1 ) (: 1 ) $branch fun
    (: n 1 > ) (: n 1 - fac n * ) $branch fun
    branch
) $fac =

# Export factorial to the host environment
(I32) (: fac ) "factorial" export
```

1.2.2 Type System

The language currently features an incomplete type system (with some compile-time-only values not fitting into the type system). However for the paper we will simplify it to exclude these values (notably namespaces, string literals, etc.) as they don't do anything interesting.

- **Primitives:** Types supported by WebAssembly: `I32 I64 F32 F64`
- **Tuples:** Types which represent the concept of adjacent values on the stack. And thus actually represent multiple values when compiled.
  - **Relevant operators:** `pack unpack`.
  - **Example:** `(F32 F32 F32) $FloatVec3 =`.
- **Unions:** Created using the `|` operator, either of it's two operands can satisfy it. At present, all union types must be resolved at compile time (ie - unions can exist as types but not values), but this should be resolved once the garbage collector is finished.
- **Unit & Void**: `Unit` is the type of the empty tuple and `Void` does not typecheck.
- `Any`: matches with any given type/value. Useful for pattern matching.
- `Arrow` **Types**: the `Arrow` operator is used for macro types. Example: `(F32 F32) (F32) Arrow $BinaryOperator =`
- **Classes:** A similar concept in other languages would likely be 'traits', adding a class to a type gives it functionality designated by that class. Thus we can define `I32 class $Color =` and then use bitwise operators in order to make the `Color` class represent a 32bit packed RGBA value. In addition to there are a number of more advanced ways to use the class
  - **Instantiating:** the `make` operator is used to add a class to a value. So for the RGBA class we can do `0xff00ff80 Color make $half_magenta =`
  - **Subclasses:** We can make a class of our color class to add new functionality, for example `Color class $RenderEngineColor =`
  - **Parametric Classes:** In addition to taking types as arguments, the class operator can take a macro, this allows one to describe more complex types. One such usage is to define type operators.
    ```
    (:  $T = (T T T) ) class $Vec3 =
    (1 2 3) I32 Vec3 make $pos =
    ```
    Note that here `class` is technically optional and the following would be the same but less strict.
    ```
    (:  $T = (T T T) ) $Vec3 =
    (1 2 3) $pos =
    ```
    so that `pos type I32 Vec3 == :data` gives 1

    In the paper we will avoid ambiguity likely take the approach of adding 'type macros' with a brackets syntax instead.
  - **Recursive Classes:** By marking the macro in the class as recursive it's only accessed as a reference to an object stored on the heap. This allows the programmer to define recursive types. Will be added once GC is stable.
    ```
    (rec:  $T = (T List T) Unit | ) class $List =
    ( 1 ( 2 () I32 List make ) I32 List make ) I32 List make
    ```

2 Syntax

Here some operations also include their stack arguments, this is intended only for clarity as to what their function is as their arguments are come from the stack when it's their turn to be evaluated, not syntactically.

$\tau ::= \mathsf{I32} \mid \mathsf{I64} \mid \mathsf{F32} \mid \mathsf{F64} \mid (\tau^*) \mid \mathsf{Unit} \mid \tau_1\ \tau_2\ \mathsf{Arrow}$
$\quad \mid\ e\ \mathsf{type} \mid e\ \mathsf{\$T} = \mid \mathtt{T} \mid \tau\ \mathsf{unpack}$

$e ::= e\ e + \mid e\ e\ \times \mid e\ e\ e\ \mathsf{select} \mid e\ e\ == \mid \tau\ \tau\ ==$
$\quad \mid\ (e^*) \mid ((\tau^*)\!: e^*\ ) \mid e\ \tau\ \mathsf{fix} \mid \overline{n}_{\mathsf{I32}} \mid \overline{n}_{\mathsf{I64}} \mid \overline{n}_{\mathsf{F32}} \mid \overline{n}_{\mathsf{F64}}$
$\quad \mid\ e\ \mathsf{\$x} = \mid \mathtt{x} \mid e\ @ \mid e\ \mathsf{unpack}$

$prog ::= e^* \mid \tau^* \mid prog\ prog \mid \varepsilon$

Where ε is used to denote an empty program/expression

2.1 Included

In addition to the following, tuples and tuple related operators, `type` operator, `Arrow` type, and some math and comparison operators were included identically to their counterparts in base language.

2.1.1 Number Literals

Number literals are denoted with subscript indicating the numeric type they correspond with. This mirrors the actual language which uses the C-style syntax for number literal type inference.

2.1.2 Branching

As mentioned before a ternary operator would be a reasonable subset of branching via function overloading. The select operator is defined as an operator which takes a condition and two values of the same type, and gives one depending on the condition. Which could be defined like so:

```
((I32 Any Any): type swap type == ) (: ( $c $a $b ) = b ) $select fun
((I32 Any Any): type swap type == && ) (: ( $c $a $b ) = a ) $select fun
```

2.1.3 Identifiers

Identifiers are included, however, it was found that the notation for dereferencing an identifier `$x` was confusing and thus the subset was changed such that unescaped identifiers are dereferenced instead of called and values must be explicitly called via the `@` operator.

2.1.4 Macros

Macros use a specific variation of the type annotations, notably where the input types are specified and the output types are inferred. In order to enable recursion, a syntactic fixed-point combinator was provided to replace the `rec` syntax.

2.2 Not included

Things which were either out of scope, problematic, or uninteresting were unincluded. Here a re some of the notable ones.

2.2.1 Classes

As much as we would like to include classes (notably recursive classes), it would likely add excess complexity, requiring subtyping (on N dimensions); untyped closures which can operate on types; and likely some other painpoints. And it's not clear how they could make the system unsafe as apart from being used to define recursive types are simply specialized versions of preexisting types.

2.2.2 Functions

Really the only reason the function overloading system was preferred instead of normal branching in the original language was in order to make it easier to extend the language. Function overloading seems like it would be needlessly painful to formalize and doesn't contribute to our proof of soundness any more than branching via a ternary. As the case for functions having potentially no possible branch results in a type error in the complete language and this is simplified by having an else clause.

2.2.3 Unions

Although the language currently supports some operations with union types, the system does not work at runtime when the true type is not known, thus, a flaw in the base language was discovered while working on the semantics for this subset and thus the original system needs to be redesigned. This would likely mean that the only way to make runtime union-types would be via the `make` operator which is related to `class` and we would additionally need to either make a different way to branch on unions or make the compiler able to determine if the condition implies a particular union member. Additionally Void was not included as on paper, it doesn't work with the type inference algorithm.

```
(: type I32 == ) (:
    # How do we know that it's an i32 here?
) $branch fun
(()(I32 F32): ... ) @ branch
```

3 Small Step Semantics For Values

3.1 Notation

To annotate stack values we used the notation of two values being written next to each other, as they would be written in the language itself. The values at the bottom of the stack (furthest to the left) are often irrelevant, and thus for most rules can be omitted with this notation. The superscript * notation (ie - e^*) is used to annotate 0 or more of the preceding.

3.2 Value and Step Judgement

The step judgement states that expressions are evaluated from left to right until everything is a value. Values are defined further as we go through.

$$\frac{e_1 \mapsto e_1' \qquad v_i \ \mathsf{val}\ \forall\ v_i \in v^*}{v^*\ e_1\ ...\ e_n \mapsto v^*\ e_1'\ ...\ e_n}\ (\textsc{Step}) \qquad \frac{\forall\ \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\overline{n}_\tau\ \mathsf{val}}\ (\text{V-}\textsc{Num})$$

$$\frac{e^* \mapsto e^{*\prime}}{(e^*) \mapsto (e^{*\prime})}\ (\text{S-}\textsc{Tuple}) \quad \frac{v_i\ \mathsf{val}\ \forall v_i \in v^*}{(v^*)\ \mathsf{val}}\ (\text{V-}\textsc{Tuple})$$

3.3 Macros, Identifiers and Scoping

The store, σ, consists of a list of scopes, where each scope is a mapping from identifiers to values. Notice the addition of an auxiliary `end_scope` operator which removes the current scope. A macro literal steps to a macro value by first stepping its input types until they're all values before substituting all of the values in the store within its body (forming a closure) (this rule comes later). Calling a macro value pushes a new empty mappings list onto the store, then places the macro's body and an end_scope operator onto the stack, enabling locals.

$$\frac{v\ \mathsf{val} \qquad \sigma = [[...], ...[x_1 \mapsto v_1\ ...\ x_n \mapsto v_n]] \qquad \sigma' = [[...], ...[x_1 \mapsto v_1\ ...\ x_n \mapsto v_n, x_{n+1} \mapsto v]]}{\sigma; v\ \$\mathsf{x} = \mapsto \sigma'; \varepsilon}\ (\textsc{S-Let})$$

$$\frac{\mathtt{x} \mapsto v \in \sigma(-1)}{\sigma; \mathtt{x} \mapsto \sigma; v}\ (\textsc{S-Ident}) \qquad \frac{\sigma' = \sigma \cup [\,] \qquad v = \mathtt{((}\tau^*\mathtt{):}\ e^*\ \mathtt{)} \qquad v\ \mathsf{val}}{\sigma; v\ \mathtt{@} \mapsto \sigma'; e^* \mathsf{end\_scope}}\ (\textsc{S-Call})$$

$$\frac{f = v\ \tau\ \mathsf{fix} \qquad f\ \mathsf{val}}{f\ \mathtt{@} \mapsto f\ v\ \mathtt{@}}\ (\textsc{S-Fix}) \qquad \frac{\sigma = [s_1, ...s_n, s_{n+1}] \qquad \sigma' = [s_1, ...s_n]}{\sigma; \mathsf{end\_scope} \mapsto \sigma'; \varepsilon}\ (\textsc{S-End})$$

3.4 Others

Notice the use of the Kronecker Delta to define the equals operator. Here τ indicates the rule is defined for all numeric types τ

$$\frac{v_i\ \mathsf{val}\ \forall\ v_i \in v^*}{(v^*)\ \mathsf{unpack} \mapsto v^*}\ (\textsc{S-Unpack}) \qquad \frac{\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\overline{n_1}_\tau\ \overline{n_2}_\tau\ +\ \mapsto \overline{(n_1 + n_2)}_\tau}\ (\textsc{S-Add})$$

$$\frac{\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\overline{n_1}_\tau\ \overline{n_2}_\tau\ \times\ \mapsto \overline{(n_1 \times n_2)}_\tau}\ (\textsc{S-Mul}) \qquad \frac{\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\overline{n_1}_\tau\ \overline{n_2}_\tau\ == \mapsto \overline{(\delta_{n_1 n_2})}_{\mathsf{I32}}}\ (\textsc{S-Eq})$$

$$\frac{v_1\ \mathsf{val} \qquad v_2\ \mathsf{val}}{\overline{0}_{\mathsf{I32}}\ v_1\ v_2\ \mathsf{select} \mapsto v_2}\ (\textsc{S-Tern-0}) \quad \frac{v_1\ \mathsf{val} \qquad v_2\ \mathsf{val} \qquad n \neq \overline{0}_{\mathsf{I32}}}{\overline{n}_{\mathsf{I32}}\ v_1\ v_2\ \mathsf{select} \mapsto v_1}\ (\textsc{S-Tern-1})$$

4 Small Step Semantics for Types

Because there are operations involving types, we have to have a equivalents to val and $\mapsto$ judgements for types. These are provided by prim and $\rightarrowtail$ respectively.

4.1 Primitives

$$\frac{\tau \in \mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}}{\tau\ \mathsf{prim}}\ (\textsc{P-Num}) \qquad \frac{\tau_1\ \mathsf{prim} \qquad \tau_2\ \mathsf{prim}}{\tau_1\ \tau_2\ \mathsf{Arrow}\ \mathsf{prim}}\ (\textsc{P-Arrow})$$

4.2 Step Judgement

Type expressions are evaluated left to right until everything is prim. Types and expressions have equal precedence and both can be present on the stack.

$$\frac{\tau \rightarrowtail \tau' \qquad \tau_i\ \mathsf{prim}\ \forall\ \tau_i \in \tau_1^*}{\tau_1^*\ \tau\ \tau_2^* \rightarrowtail \tau_1^*\ \tau'\ \tau_2^*}\ (\textsc{Ty-Step}) \qquad \frac{\tau^* \rightarrowtail \tau^{*\prime} \qquad v_i\ \mathsf{val}\ \forall v_i \in v^*}{v^*\ \tau^*\ e^*\ \rightarrowtail v^*\ \tau^{*\prime}\ e^*}\ (\textsc{Ty-Step-Mixed})$$

$$\frac{e^* \mapsto e^{*\prime} \qquad \tau_i\ \mathsf{prim}\ \forall\ \tau_i \in \tau^*}{\tau^*\ e^*\ \tau_2^* \mapsto \tau^*\ e^{*\prime}\ \tau_1}\ (\textsc{Val-Step-Mixed})$$

4.3 Type Identifiers

Type identifiers use the same store as used for values, however their identifiers are guaranteed to not conflict as type identifiers start with a capital letter vs lower-case for value identifiers. Because we use the same store the step semantics for closures and scopes continue to work.

$$\frac{\sigma = [[...], ...[T_1 \rightarrowtail \tau_1 ... T_n \rightarrowtail \tau_n]] \qquad \begin{array}{c}\tau\ \mathsf{prim}\\ \sigma' = [[...], ...[T_1 \rightarrowtail \tau_1 ... T_n \rightarrowtail \tau_n, T_{n+1} \rightarrowtail \tau]]\end{array}}{\sigma; \tau\ \texttt{\$T =} \rightarrowtail \sigma'; \varepsilon}\ (\textsc{Ty-Let})$$

$$\frac{\sigma = [..., [x_1 \mapsto v_1\ ...\ x_n \mapsto v_n,\ T_1 \rightarrowtail \tau_1\ ...\ T_n \rightarrowtail \tau_n]] \qquad \tau_i\ \mathsf{val}\ \forall\ \tau_i \in \tau^*}{\sigma; \texttt{((}\tau^*\texttt{):}\ e^*\ \texttt{)} \mapsto \sigma; \texttt{((}\tau^*\texttt{):}\ [v_1/x_1]...[v_n/x_n][\tau_1/T_1]...[\tau_n/T_n]e^*\ \texttt{)}\ \mathsf{val}}\ (\textsc{Macro-Subst})$$

$$\frac{\texttt{T} \mapsto \tau \in \sigma(-1)}{\sigma; \texttt{T} \rightarrowtail \sigma; \tau}\ (\textsc{Ty-Ident})$$

4.4 Tuples

$$\frac{\tau^* \rightarrowtail \tau^{*\prime}}{(\tau^*) \rightarrowtail (\tau^{*\prime})}\ (\textsc{Ty-Tuple}) \qquad \frac{\tau_i\ \mathsf{prim}\ \forall \tau_i \in \tau^*}{(\tau^*)\ \mathsf{prim}}\ (\textsc{P-Tuple}) \qquad \frac{\tau_i\ \mathsf{prim}\ \forall\ \tau_i \in \tau^*}{(\tau^*)\ \mathsf{unpack} \rightarrowtail \tau_1 ... \tau_n}\ (\textsc{Ty-Unpack})$$

4.5 Others

$$\frac{\tau_1\ \mathsf{prim} \qquad \tau_2\ \mathsf{prim}}{\tau_1\ \tau_2\ \ \texttt{==} \mapsto\ \overline{(\delta_{\tau_1 \tau_2})}_{\mathsf{I32}}}\ (\textsc{S-TyEq}) \qquad \frac{e\ \mathsf{val} \qquad \tau_1\ \mathsf{prim}}{e\ \tau_1\ \mathsf{fix\ val}}\ (\textsc{V-Fix})$$

$$\frac{\tau^* \rightarrowtail \tau^{*\prime}}{\sigma; \texttt{((}\tau^*\texttt{):}\ e^*\ \texttt{)} \rightarrowtail \sigma; \texttt{((}\tau^{*\prime}\texttt{):}\ e^*\ \texttt{)}}\ (\textsc{Ty-Macro}) \qquad \frac{v\ \mathsf{val} \qquad v : \tau}{v\ \mathsf{type} \rightarrowtail \tau}\ (\textsc{Ty-Typeof})$$

5 Typing

Note that we're using a somewhat weird notation for a stack machine (normally there's arrows representing stack transformation). This aligns with how we defined our syntax.

5.1 Step Judgement

$$\frac{\Gamma \vdash e : \tau \qquad \tau \rightarrowtail \tau'}{\Gamma \vdash e : \tau'}\ (\textsc{T-Step})$$

5.2 Math and Logic

$$\frac{\tau \in \mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}}{\Gamma \vdash \overline{n}_\tau : \tau}\ (\textsc{T-Num}) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\Gamma \vdash e_1\ e_2\ + : \tau}\ (\textsc{T-Add})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\Gamma \vdash e_1\ e_2\ \times : \tau}\ (\textsc{T-Mul})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{I32} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ e_2\ e_3\ \mathsf{select} : \tau}\ (\textsc{T-Tern})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\Gamma \vdash e_1\ e_2\ == : \mathsf{I32}}\ (\textsc{T-Eq}) \qquad \frac{\tau_1\ \mathsf{prim} \quad \tau_2\ \mathsf{prim}}{\Gamma \vdash \tau_1\ \tau_2\ == : \mathsf{I32}}\ (\textsc{T-TyEq})$$

5.3 Tuples

$$\frac{}{\Gamma \vdash () : \mathsf{Unit}}\ (\textsc{T-Unit}) \qquad \frac{\tau^* = \{\tau_i \text{ such that } \Gamma \vdash e_i : \tau_i\ \forall\ e_i \in e^*\}}{\Gamma \vdash (e^*) : (\tau^*)}\ (\textsc{T-Tuple})$$

$$\frac{\Gamma \vdash (e^*) : (\tau^*)}{\Gamma \vdash (e^*)\ \mathsf{unpack} : \tau^*}\ (\textsc{T-Unpack})$$

5.4 Macros

This is why we needed the step judgement

$$\frac{\tau^* = \tau_1\ ...\ \tau_n \quad v^* = v_1 : \tau_1\ ...\ v_n : \tau_n \quad m = \texttt{((}\tau^*\texttt{):}\ e^*\ \texttt{)} \quad m\ \mathsf{val}}{\Gamma \vdash \texttt{((}\tau^*\texttt{):}\ e^*\ \texttt{)} : (\tau^*)\ (v^*\ m\ \texttt{@})\ \mathsf{type\ Arrow}}\ (\textsc{T-Macro})$$

$$\frac{\Gamma \vdash e_1 : (\tau^*)\ (\tau^{*\prime})\ \mathsf{Arrow} \quad \Gamma \vdash e^* : \tau^*}{\Gamma \vdash e^*\ e_1\ \texttt{@} : \tau^{*\prime}}\ (\textsc{T-Call}) \qquad \frac{\Gamma(\texttt{x}) = \tau}{\Gamma \vdash \texttt{x} : \tau}\ (\textsc{T-Ident})$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(\texttt{x}) = \tau}{\Gamma \vdash e\ \texttt{x} =: \varepsilon}\ (\textsc{T-Let})$$

$$\frac{\tau_1\ \mathsf{prim} \quad \tau_1 = (\tau_3^*)\ (\tau_4^*)\ \mathsf{Arrow} \quad \Gamma \vdash e : (\tau_3^*\ \tau_1)\ (\tau_4^*)\ \mathsf{Arrow}}{\Gamma \vdash e\ \tau_1\ \mathsf{fix} : \tau_1}\ (\textsc{T-Fix})$$

$$\frac{}{\Gamma \vdash \mathsf{end\_scope} : \varepsilon}\ (\textsc{T-End})$$

6 Proofs

6.1 Theorem - Progress

If $e^* : \tau^*$ then either $e^* \mapsto e^{*\prime}$ or e_i val $\forall\ e_i \in e^*$.

Proof. By induction on the derivation of $e^* : \tau^*$.

- (T-Num) Then $e^* = \overline{n}_\tau$, where $\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}$. By (V-Num), $\overline{n}_\tau$ val, so e^* val.

- (T-Add) Then $e^* = e_1\ e_2\ +$ and $e_1 : \tau$ and $e_2 : \tau$ for some $\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}$.

  By induction, $e_1 \mapsto e_1'$ or e_1 val. If $e_1 \mapsto e_1'$, then by (Step), $e^* \mapsto e_1'\ e_2\ +$.

  If e_1 val, then by induction, $e_2 \mapsto e_2'$ or e_2 val. If e_1 val and $e_2 \mapsto e_2'$, then by (Step), $e^* \mapsto e_1\ e_2'\ +$.

  If e_1 val and e_2 val, then by CF, $e_1 = \overline{n_1}_\tau, e_2 = \overline{n_2}_\tau$, and by (S-Add), $e^* \mapsto \overline{n_1 + n_2}_\tau$.

- (T-Mul) $[\times/+]$[(S-Mul)/(S-Add)] (T-Add)

- (T-Tern) Then $e^* = e_1\ e_2\ e_3$ select and $e_1 : \mathsf{I32}$.

  By induction, $e_1 \mapsto e_1'$ or e_1 val. If $e_1 \mapsto e_1'$, then by (Step), $e^* \mapsto e_1'\ e_2\ e_3$ select.

  If e_1 val, then by induction, $e_2 \mapsto e_2'$ or e_2 val. If e_1 val and $e_2 \mapsto e_2'$, then by (Step), $e^* \mapsto e_1\ e_2'\ e_3$ select.

  If e_1 val and e_2 val, then by induction, $e_3 \mapsto e_3'$ or e_3 val. If e_1 val and e_2 val and $e_3 \mapsto e_3'$, then by (Step), $e \mapsto e_1\ e_2\ e_3'$ select.

  If e_1 val and e_2 val and e_3 val, then by CF, $e_1 = \overline{n}_{\mathsf{I32}}$.

  If $n = 0$, then by (S-Tern-0), $e^* \mapsto e_3$. Otherwise, $n \neq 0$, so by (S-Tern-1), $e^* \mapsto e_2$.

- (T-Unit) Then $e^* = ()$. All zero of the tuple's elements are values, so by (V-Tuples), () val.

- (T-Tuple) Then $e^* = (e_t^*) = (e_1\ e_2\ \ldots\ e_n)$.

  By induction, $\forall e_i \in e_t^*$, e_i val or $e_i \mapsto e_i'$.

  If all elements are values, then (e_t^*) val by (V-Tuple).

  Otherwise, one or more elements can step, including a "leftmost steppable element" $e_j \mapsto e_j'$. Then, by (Step), $e_t^* \mapsto [e_j'/e_j]e_t^*$, so by (S-Tuple), $e^* \mapsto ([e_j'/e_j]e_t^*)$.

- (T-Unpack) Then $e^* = (e_t^*)$ unpack. By induction, either $(e_t^*) \mapsto (e_t^{*'})$ or (e_t^*) val.

  If $(e_t^{*'}) \mapsto (e_t^{*'})$, then by (Step), (e_t^*) unpack $\mapsto (e_t^{*'})$ unpack.

  If (e_t^*) val, then by (S-Unpack), (e_t^*) unpack $\mapsto e_t^*$.

- (T-Macro) Then $e^* = m =$ `((`τ^*`):` e_t^* `)`. m val by the premise. A macro isn't well-typed if it's not a value, and it's not a value if it hasn't had its free variables bound using (Macro-Subst).

- (T-Call) Then $e^* = e_x^*\ e_f$ **@**. By induction, either e_i val $\forall\ e_i \in e_x^*$ or $e_x^* \mapsto e_x^{*'}$.

  If $e_x^* \mapsto e_x^{*'}$, then by (Step), $e^* \mapsto e_x^{*'}\ e_f$ **@**.

  Otherwise, e_i val $\forall\ e_i \in e_x^*$ and, by induction, e_f val or $e_f \mapsto e_f'$.

  If $e_f \mapsto e_f'$, then by (Step), $e^* \mapsto e_x^*\ e_f'$ **@**.

  Otherwise, e_f val. By CF, e_f is either a macro or an application of fix. If it's a macro with a body e_b^*, then by (S-Call), $e^* \mapsto e_b^*$ **end_scope**. If it's an application of fix to some f, then by (S-Fix), $e^* \mapsto e_x^*\ e_f\ f$ **@**.

- (T-Ident) Then $e^* =$ `x` and some binding `x` $\mapsto v$ is in scope. By (S-Ident), $e^* \mapsto v$.

- (T-Let) Then $e^* = e$ `$x =`. By induction, e val or $e \mapsto e'$. If $e \mapsto e'$, then by (Step), $e^* \mapsto e'$ `$x =`. If e val, then by (S-Let), $\sigma; e^* \mapsto \sigma'; \varepsilon$, where σ' is, as described in the rule, σ with `x` $\mapsto e$ added to its "topmost" scope.

- (T-Fix) Then $e^* = e\ \tau_1$ fix where $e : (\tau_{in}^*\ \tau_1)\ \tau_{out}$ Arrow and τ_1 prim. By induction, e val or $e \mapsto e'$. If $e \mapsto e'$, then by (Step), $e^* \mapsto e'\ \tau_1$ fix. If e val, then by (V-Fix), e^* val.

- (T-End) Lemma: The presence of an `end_scope` operator on the stack to execute implies the presence of a scope in the store to pop.
  Proof: `end_scope` is absent from the formal syntax, so it can only exist if it is introduced during execution. It is introduced exclusively via (S-Call), which pushes exactly one scope, and eliminated exclusively via (S-End), which pops exactly one scope.

  Then $e^* =$ `end_scope`, and by the lemma, there is a scope to pop. Then by (S-End), $e^* \mapsto \varepsilon$.

$\square$

6.2 Lemma - Primitive Values

If v val and $v \vdash \tau$:, then τ prim.

Proof. By induction on the derivation of v val.

- (V-Num) Then $v = \overline{n}_\tau$ and $\tau \in \mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}$. By (P-Num), τ prim.
- (V-Tuple) Then $v = (v^*)$ and $\tau = (\tau_i^*)$ where $v_i : \tau_i$ and v_i val $\forall v_i \in v^*$. By induction, τ_i prim $\forall \tau_i \in \tau^*$. So by (P-Tuple), τ prim.
- (Macro-Subst) Then v is a macro with its captures substituted in. By (T-Macro), τ is of the form τ_1 τ_2 Arrow. By (P-Arrow), τ prim.
- (V-Fix) Then $v = m$ τ_1 fix where $t_1 = \tau_3^*$ τ_4^* Arrow and m : `((`t_3^* t_1`):` t_4^* `)`. Then $v : \tau_1$ by (T-Fix), meaning $\tau = \tau_1$, so by (P-Arrow), τ prim.

□

6.2.1 Lemma - Convergence

[1] Part 1: $\tau^* \rightarrowtail \tau^{*\prime}$ is no longer possible when τ_i prim $\forall$ $\tau_i \in \tau^*$.

Proof. By induction on the derivation of τ prim.

- (P-Num) Then $\tau \in \mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}$. By exhaustion, there is no rule of the form $\tau \rightarrowtail \tau'$ that matches.
- (P-Arrow) Then $\tau = \tau_1$ τ_2 Arrow and τ_1 prim and τ_2 prim. By (Ty-Step), τ can step if τ_1 can step or if τ_1 prim and τ_2 can step, but by induction, neither τ_1 nor τ_2 can step. By exhaustion, there is no other rule that would allow τ to step.
- (P-Tuple) Then $\tau = (\tau^*)$ where τ_i prim $\forall \tau_i \in \tau^*$. By (Ty-Tuple), a tuple type can step if its elements collectively can. By induction, they cannot. By exhaustion, there is no other rule that would allow τ to step.

□

Part 2: $\tau^* \rightarrowtail \tau^{*\prime}$ will eventually step to a $\tau^{*\prime}$ such that τ_i prim $\forall$ $\tau_i \in \tau^{*\prime}$.

Proof. This is provable by the fact that the language does not feature recursive types and the only form of recursion in the language is via the fix operator and type operator does not have to follow recursive paths in order to get its type (Type Rules (T-Macro), (T-Fix)). The other rules are trivial (i.e., step to a prim) and/or irrelevant to the assertion. □

6.3 Theorem - Preservation

If $\Gamma \vdash e^* : \tau^*$ and $e^* \mapsto e^{*\prime}$ then $\Gamma \vdash e^{*\prime} : \tau^{*\prime}$ where $\tau^* \rightarrowtail^* \tau^{*\prime}$.

Proof. By induction on the derivation of $e^* \mapsto e^{*\prime}$.

- (Step) Then $e^* = v^* e_1 ... e_n$ and $e^* \mapsto v^* e_1' ... e_n$ where $e_1 : \tau_1$ and $e_1' : \tau_1'$ and v_i val $\forall v_i \in v^*$. By induction, $\tau_1 \rightarrowtail^* \tau_1'$. By the Primitive Values lemma, the types of those values on the left are all primitive, so by (Ty-Step), $\tau^* \rightarrowtail^* [\tau_1'/\tau_1]\tau^* = \tau^{*\prime}$.
- (S-Tuple) Then $e^* = (e^*_{elem})$ and $e^* \mapsto (e^{*\prime}_{elem})$ and $\tau^* = (\tau^*_{elem})$. By induction, $e^{*\prime}_{elem} : \tau^{*\prime}_{elem}$ where $\tau^*_{elem} \rightarrowtail^* \tau^{*\prime}_{elem}$. By (T-Tuple), $(e^{*\prime}_{elem}) : (\tau^{*'}_{elem})$, and by (Ty-Tuple), $(\tau^*_{elem}) \rightarrowtail (\tau^{*'}_{elem})$.
- (S-Let) Then $e^* = v$ `$x` $=$. By (T-Let), $e^* : \varepsilon$, so $\tau^* = \varepsilon$. $e^* \mapsto \varepsilon$, so $\tau^* = \tau^{*\prime} = \varepsilon$.
- (S-Ident) Then $e^* =$ `x` and some binding `x` $\mapsto v$ is in scope where $v : \tau_v$. $e^* \mapsto v$ and, by (T-Ident), $e^* : \tau_v$, so $\tau^* = \tau^{*\prime} = \tau_v$.

[1]a.k.a. Progress for Types, kinda

- (S-Call) Then $e^* = m$ @, where $m =$ `((`τ^*_{in}`):` e^* `)` and m val.

  This is going to get a little weird.

  First, (Macro-Subst) means that m val iff all its free variables have been replaced with captures.

  Next, (T-Macro) states that once that's done, a macro is well-typed only if it's "ready to be called" (appropriate arguments are on the stack), because its return type is "whatever you get when you call it", i.e., we're doing some kind of type inference because we tried too hard. e^* being well-typed is part of the theorem's premise, so the arguments are indeed in place, and we can call them v^* (with types τ^*_{in}, of course).

  To get the return type, we start with the (v^* m @) type described in (T-Macro). By the Convergence lemma, we can step this type-expression to a primitive, and by (Ty-Typeof) and (T-Tuple), the result will be a tuple type, which we can name (τ^*_{out}).

  That then gives us m : (τ^*_{in}) (τ) Arrow. As previously established, the necessary arguments $v^* : \tau^*_{in}$ are on the stack, so by (T-Call), $e^* : \tau^*_{out}$.

  e^*, a macro call, steps to the macro's expansion (i.e., its body plus an `end_scope`). The "type signature" of the macro's expansion steps to τ^*_{out}. The type of e^* also steps to τ^*_{out}. That's preservation.

- (S-Fix) Then $e^* = f$ @ where $f = v\ \tau$ fix and $e^* \mapsto e^{*\prime} f\ v$ @.

  By (T-Call), the type of $e^{*\prime}$ is inferred[2] based on the expansion of v. By (T-Fix), this is also the case for e^*.

- (S-End) Then $e^* =$ `end_scope`. By (T-End), $e^* : \varepsilon$, so $\tau^* = \varepsilon$. $e^* \mapsto \varepsilon$, so $\tau^* = \tau^{*\prime} = \varepsilon$.

- (S-Unpack) Then $e^* = (v^*)$ unpack, where $v^* : \tau^*_{elem}$. By (T-Unpack), $\tau^* = \tau^*_{elem}$. $e^* \mapsto v^*$, so $\tau^{*\prime} = \tau^* = \tau^*_{elem}$.

- (S-Add) Then $e^* = \overline{n_1}_\tau\ \overline{n_2}_\tau\ +$ and $e^* \mapsto \overline{n_1 + n_2}_\tau$ for some $\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}$. By (T-Add), $\overline{n_1}_\tau\ \overline{n_2}_\tau\ + : \tau$, and by the appropriate choice of (T-Int), (T-Long), (T-Float), or (T-Double), $v_3 : \tau$, so $\tau^* = \tau^{*\prime} = \tau$.

- (S-Mul) Omitted for brevity on the grounds of being mechanically identical to the above case.

- (S-Eq) Then $e^* = \overline{n_1}_\tau\ \overline{n_2}_\tau\ ==$ and $e^* \mapsto \overline{(\delta_{n_1 n_2})}_{\mathsf{I32}}$ for some $\tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}$. By (T-Eq), $\tau^* = \mathsf{I32}$. By (T-Int), $\overline{(\delta_{n_1 n_2})}_{\mathsf{I32}} : \mathsf{I32}$, so $\tau^{*\prime} = \tau^* = \mathsf{I32}$.

- (S-Tern-0), (S-Tern-1) Then $e^* = \overline{n}_{\mathsf{I32}}\ v_1\ v_2$ select and $v_1 : \tau$ and $v_2 : \tau$ for some τ. By (T-Tern), $e^* : \tau$, so $\tau^* = \tau$. Either $e^* \mapsto v_1$ or $e_* \mapsto v_2$, so in either case, $\tau^* = \tau^{*\prime} = \tau$.

- (Macro-Subst) Then e^* is a macro with free variables that steps to a macro with captures. The fact that macros perform type inference based on the stack with which they're called (which isn't really known in the context of this rule) makes things pretty complicated, but it should still be the case that by the substitution lemma, this capturing changes nothing.

$\square$

7 Conclusion

As mentioned before the big takeaway was that we learned some flaws with the base language, notably that union types need to be redesigned. The semantics for the language are overly complicated, and while working on proofs we came up with some ideas for simplifying them further. We made light use of the lemmas for canonical forms and substitution, but for the former the cases were fairly trivial anyway, and for the latter, the proof case (and the rule it concerns) is somewhat weird and handwavy even with the lemma.

[2] I'm tired.