Type Safety

Informally, a type-safe language is one for which

- There is a clearly specified notion of type correctness.
- Type correct programs are free of “run-time type errors”.

But this begs the question!

Type Safety

Type safety is a matter of coherence between the static and dynamic semantics.

- The static semantics makes predictions about the execution behavior.
- The dynamic semantics must comply with those predictions.

What is a run-time type error?

- Bus error?
- Division by zero? Arithmetic overflow?
- Array bounds check?
- Uncaught exception?

For example, if the type system tracks sizes of arrays, then out-of-bounds subscript is a run-time type error.

- The type system ensures that access is within allowable limits.
- If the run-time model exceeds these bounds, you have a run-time type error.

Similarly, if the type system tracks value ranges, then division by zero or arithmetic overflow is a run-time type error.
Type Safety

Demonstrating that a program is well-typed means proving a theorem about its behavior.

- A type checker is therefore a theorem prover.
- Non-computability theorems limit the strength of theorems that a mechanical type checker can prove.

Fundamentally there is a tension between
- the expressiveness of the type system, and
- the difficulty of proving that a program is well-typed.
Therein lies the art of type system design.

Formalization of Type Safety

The coherence of the static and dynamic semantics is neatly summarized by two related properties:

1. Preservation. Well-typed programs do not “go off into the weeds”. A well-typed program remains well-typed during execution.
2. Progress. Well-typed programs do not “get stuck”. If an expression is well-typed, then either it is a value or there is a well-defined next instruction.

More precisely, type safety is the conjunction of two properties:

1. Preservation. If $e : r$, and $e \rightarrow e'$, then $e' : r$.
2. Progress. If $e : r$, then either $e \text{ val}$, or there exists $e''$ such that $e \rightarrow e''$.

Consequently, if $e : r$ and $e \rightarrow^* v$, then $v : r$.

Proof of Preservation for $L[\text{num str}]$

Theorem 1 (Preservation)
If $e : r$ and $e \rightarrow e'$, then $e' : r$.

Proof: The proof proceeds by induction on evaluation, that is, induction on the transition judgement. This means

1. We must prove it outright for axioms (rules with no premises).
2. For each rule, we may assume the theorem for the premises, and show it is true for the conclusion.

Thus if $e : \text{num}$ and $e \rightarrow^* v$, then $v = \text{num}[n]$ for some $n$. In words: expressions of type $\text{num}$ evaluate to natural numbers.

Formalization of Type Safety
Proof of Preservation for $\mathcal{L}(\text{num str})$ Instruction Steps

The primitive operations are straightforward:

We have $\epsilon = \text{plus}(\text{num}[n_1], \text{num}[n_2])$, $\tau = \text{num}$, and $\epsilon' = \text{num}[n_1 + n_2]$.

Clearly $\epsilon' : \text{num}$, as required.

The cases for multiplication, concatenation and length are handled similarly.

Proof of Preservation for $\mathcal{L}(\text{num str})$ Search Rules

Consider the search rule:

$$\epsilon_1 \Rightarrow \epsilon'_1$$

$$\text{plus}(\epsilon_1, \epsilon_2) \Rightarrow \text{plus}(\epsilon'_1, \epsilon'_2)$$

We have $\epsilon = \text{plus}(\epsilon_1, \epsilon_2)$, $\epsilon' = \text{plus}(\epsilon'_1, \epsilon'_2)$, and $\epsilon_1 \Rightarrow \epsilon'_1$.

By inversion $\epsilon_1 : \text{num}$ and $\epsilon_2 : \text{num}$. By induction $\epsilon'_1 : \text{num}$, and hence $\epsilon' : \text{num}$, as required. The case for the similar rule for multiplication is handled similarly.

Proof of Preservation for $\mathcal{L}(\text{num str})$ Search Rules

Consider the search rule:

$$\epsilon_1 \Rightarrow \epsilon'_1$$

$$\text{cat}(\epsilon_1, \epsilon_2) \Rightarrow \text{cat}(\epsilon'_1, \epsilon'_2)$$

We have $\epsilon = \text{cat}(\epsilon_1, \epsilon_2)$, $\epsilon' = \text{cat}(\epsilon'_1, \epsilon'_2)$, and $\epsilon_1 \Rightarrow \epsilon'_1$.

By inversion $\epsilon_1 : \text{str}$ and $\epsilon_2 : \text{str}$. By induction $\epsilon'_1 : \text{str}$, and hence $\epsilon' : \text{str}$, as required.
Proof of Preservation for $L\{\text{num str}\}$ Search Rules

There is just one case for the "by value" interpretation of let expressions. Consider the search rule:

$$e_1 \mapsto e'_1$$

$$\text{let}(e_1; x. e_2) \mapsto \text{let}(e'_1; x. e_2)$$

We have $e = \text{let}(e_1; x. e_2) : \tau$, $e' = \text{let}(e'_1; x. e_2)$ and $e_1 \mapsto e'_1$.

By inversion $e_1 : \tau_1$, for some type $\tau_1$ such that $x : \tau_1 \vdash e_2 : \tau$.

By induction $\tau'_1 : \tau_1$, and hence $e' : \tau$.

Proof of Preservation for $L\{\text{num str}\}$

This completes the proof. How might it have failed?

Only if some instruction is mis-defined. For example, if we had defined

$$\text{plus}(\text{num}[0]; \text{num}[0]) \mapsto \begin{cases} \text{str}[\text{zero}] & \text{if } m = n = 0 \\ \text{num}[m + n] & \text{otherwise} \end{cases}$$

Then preservation would fail.

In other words, preservation says that the steps of evaluation are well-behaved.

Canonical Forms Lemma

The type system for $L\{\text{num str}\}$ predicts the forms of values:

Lemma 2 (Canonical Forms for $L\{\text{num str}\}$)

Suppose that $e : \tau$ and $e \text{ val}$.

1. If $\tau = \text{str}$, then $e = \text{str}[s]$ for some $s$.

2. If $\tau = \text{num}$, then $e = \text{num}[n]$ for some $n$.

Proof of Canonical Forms Lemma

The proof is by induction on typing. For example, for $e : \text{str}$,

- $e$ cannot be a natural number, because $\text{num} \neq \text{str}$.

- $e$ cannot be a variable, because it is closed.

- $e$ can be a string constant, as specified.

- $e$ cannot be an application of a primitive operation, nor a let expression.

Proof of Progress for $L\{\text{num str}\}$

Theorem 3 (Progress)

If $e : \tau$, then either $e$ is a value, or there exists $e'$ such that $e \mapsto e'$.

Proof: The proof is by induction on typing. We consider each typing rule in turn. For axioms, we must demonstrate the theorem directly. Otherwise, for each rule, we assume the theorem for the premises, and show it holds for the conclusion. ■
Proof of Progress for $L(\text{num \; str})$

The expression cannot be a variable, because it is closed.

For natural numbers or string constants the result is immediate because they are values.

Consider the rule for typing addition expressions. We have $e = \text{plus}(e_1; e_2)$ and $\tau = \text{num}$, with $e_1 : \text{num}$ and $e_2 : \text{num}$.

By induction we have either $e_1$ is a value, or there exists $e'_1$ such that $e_1 \mapsto e'_1$ for some expression $e'_1$.

We consider these two cases in turn.

Proof of Progress for $L(\text{num \; str})$

Suppose that $e = \lambda x (e_1; x e_2)$. Consider the case for the "by value" interpretation:

By the inductive hypothesis, either $e_1$ is a value, or there exists $e'_1$ such that $e_1 \mapsto e'_1$.

If $e_1$ is not a value, then $e \mapsto \lambda x (e'_1; x e_2)$ by the search rule for let expressions, as required.

If $e_1$ is a value, then by the inversion for typing lemma $e_1 : \tau_1$ such that $x : e_1 \mapsto \tau$ and $e \mapsto e'$, where $e' = [x_1/x]e_2$, by the rule for executing let expressions.

Proof of Progress for $L(\text{num \; str})$

If $e_1 \mapsto e'_1$, then $e \mapsto e'$, where $e' = \text{plus}(e'_1; e_2)$, which completes this case.

If $e_1$ is a value, then we note that by the canonical forms lemma $e_1 = \text{num}(n_1)$ for some $n_1$, and we consider $e_2$.

By induction either $e_2$ is a value, or $e_2 \mapsto e'_2$. If $e_2$ is a value, then by the canonical forms lemma $e_2 = \text{num}(n_2)$ for some $n_2$, and we note that $e \mapsto e'$, where $e' = \text{num}(n_1 + n_2)$.

If $e_2$ is not a value, then $e \mapsto e'$, where $e' = \text{plus}(e'_1; e'_2)$.

Extending the $L(\text{num \; str})$ Language

We deliberately omitted division from $L(\text{num \; str})$. Suppose we add div as a primitive operation and define the following evaluation rules for it:

\[
\begin{align*}
\text{div}(\text{num}(n_1); \text{num}(n_2)) & \Rightarrow \text{num}(n_1 \div n_2) \\
\text{div}(e_1; e_2) & \Rightarrow \text{div}(e'_1; e_2) \\
\text{div}(e_1; e_2) & \Rightarrow \text{div}(e_1; e'_2) \\
\text{div}(\text{val}; e_2) & \Rightarrow \text{div}(e_1; e'_2) \\
\end{align*}
\]

Proof of Progress for $L(\text{num \; str})$

The other cases are handled similarly. How could the proof have failed?

1. Some instruction step was omitted. If there were no instructions for $\text{plus}(\text{num}(n_1); \text{num}(n_2))$, then progress would fail.

2. Some search rule was omitted. If there were no rule for, say, $\text{div}(e_1; e_2)$, where $e_1$ is not a value, then we cannot make progress.

In other words, progress implies that we cannot find ourselves in an embarrassing situation!

Extending the $L(\text{num \; str})$ Language

Suppose the static semantics gives the following typing to div:

\[
\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num} \\
\Gamma \vdash \text{div}(e_1; e_2) : \text{num}
\]

Is the language still safe?

• Preservation continues to hold: new instruction preserves type.

• Progress fails: $\text{div}(\text{num}(10); \text{num}(0)) \downarrow$, yet has type num.
Extending the Language

How can we recover safety?

1. Strengthen the type system to rule out the offending case.

2. Change the dynamic semantics to avoid getting “stuck” when the denominator is zero.

Extending the Type System

A natural idea: add a type alias of non-zero integers. Revise the typing rule for division to:

\[
\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \text{div}(e_1, e_2) : \text{num}}
\]

But how do we “create” expressions of type nzint?

• This type does not have good closure properties, e.g. is not closed under subtraction.
• It is undecidable in general whether \( e : \text{num} \) evaluates to a non-zero integer.

Modifying the Dynamic Semantics

One idea: an inductively defined judgement, \( e \text{ err} \), stating that expression \( e \) incurred a checked error, such as zero denominator or array index out of bounds, at run time.

• Add rules to detect run-time errors
• Add rules to propagate errors through program execution
• Revise statement of safety to account for errors. A program has an answer that is either a value or an error.

Adding Errors

For example, we add a judgement for handling zero divisor:

\[
\begin{align*}
\text{div} & (\text{num}[1], \text{num}[0]) \Rightarrow \text{err} \\
\text{val} & \Rightarrow \text{err} \\
\text{plus} & (\text{err}, \text{err}) \Rightarrow \text{err}
\end{align*}
\]

Then we must propagate errors upwards:

\[
\begin{align*}
\text{div} & (\text{val}[1], \text{err}) \Rightarrow \text{err} \\
\text{val} & \Rightarrow \text{err} \\
\text{plus} & (\text{val}[1], \text{err}) \Rightarrow \text{err}
\end{align*}
\]

and so on for the other non-value expression forms.

Proving Progress

Theorem 4 (Progress With Error)

If \( e : \tau \), then either \( e \text{ err} \), or \( e \text{ val} \) or there exists \( e' \) such that \( e \rightarrow e' \).

Proof: The proof is by induction on typing and proceeds much as before, except that there are now three cases to consider at each point in the proof.

Modifying the Dynamic Semantics

Adding a separate set of evaluation rules to check for errors seems unnatural. An alternative is to fold error checking into evaluation.

• Introduce a special expression, \( e \text{ err} \), that signals a checked error has arisen, such as zero denominator or array index out of bounds.
• Revise typing, dynamic semantics and statement of safety to account for errors.
Adding Errors

Since it aborts computation, static semantics assigns an arbitrary type to `error`:

```
error
```

The dynamic semantics must be modified in two ways:

- Primitive operations must yield an error in an otherwise undefined state.
- Search rules must propagate errors once they arise.

For example, we add an error transition for zero divisor:

```
c1 val
div(c1, num[0]) ==> error
```

Then we must propagate errors upwards:

```
plus(error, c2) ==> error
plus(c1, error) ==> error
```

and so on for the other non-value expression forms.

Defining `e` to hold exactly when `e = error`, the Progress With Error theorem still holds.

Summary

- Type safety expresses the **coherence** of the static and dynamic semantics.
- Coherence is elegantly expressed as the conjunction of **preservation** and **progress**.

Checked errors ensure that behavior is well-defined, even in the presence of undefined operations.

- Explicitly circumscribe error transitions.
- Explicitly define which states lead to an error.