Programming Languages

MinML: A MINiMaL Functional Language

Type Safety

David Walker

Type Safety

Java and ML are **type safe**, or **strongly typed**, languages.

C and C++ are often described as **weakly typed** languages.

What does this mean?

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

What is a run-time type error?

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

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.

Type Safety

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 it's behavior.

- A type checker is therefore a **theorem prover**.

- Non-computability theorems limit the strength of theorems that a **mechanical** type checker can prove.

Type Safety

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.

Type Safety

Two common misconceptions:

- The expressiveness of type systems is inherently limited to decidable properties.

- Anything that a type checker can do can also be done at run-time (perhaps at some small cost).

Type Safety

These are both false!

- There is **no inherent limit** to the expressiveness of a type system.

- Type systems can capture **undecidable** properties such as "this function will terminate".

We will develop these ideas further as we proceed in the course.

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.

Formalization of Type Safety

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

1. **Preservation**. If $e : \tau$, and $e \mapsto e'$, then $e' : \tau$.

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

Consequently, if $e : \tau$ and $e \mapsto^* v$, then $v : \tau$.

Formalization of Type Safety

Moreover, the type of a (closed) value determines its form. If $v : \tau$, then

- If $\tau = \texttt{int}$, then $v = n$ for some n.
- If $\tau = \texttt{bool}$, then $v = \texttt{true}$ or $v = \texttt{false}$.
- If $\tau = \tau_1 \to \tau_2$, then $v = \texttt{fun}\, f\, (x\texttt{:}\tau_1)\texttt{:}\tau_2 = e$ for some f, x, and e.

Thus if $e : \texttt{int}$ and $e \mapsto^* v$, then $v = n$ for some n. In words: expressions of type `int` evaluate to numbers.

Proof of Preservation

Theorem 1 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Proof: The proof proceeds by ? 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.

$\blacksquare$

Proof of Preservation for Instruction Steps

The primitive operations are straightforward:

We have $e = \texttt{+}(n_1, n_2)$, $\tau = \texttt{int}$, and $e' = n_1 + n_2$.

Clearly $e' : \texttt{int}$, as required.

The other primitive operations are handled similarly.

Proof of Preservation for Instruction Steps

There are two cases for conditionals:

1. We have $e = \mathtt{if}_\tau\,\mathtt{true}\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2\,\mathtt{fi}$ and $e' = e_1$.

   Since $e : \tau$, we have $e_1 : \tau$, by inversion.

2. We have $e = \mathtt{if}_\tau\,\mathtt{false}\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2\,\mathtt{fi}$ and $e' = e_2$.

   Since $e : \tau$, we have $e_2 : \tau$, by inversion.

Proof of Preservation for Instruction Steps

Application is a bit more complex. We require both the inversion and the substitution lemmas.

We have $e = \texttt{apply}(v_1, v_2)$,
where $v_1 = \texttt{fun}\, f\ (x\texttt{:}\tau_2)\texttt{:}\tau\ =\ e_2$,
and $e' = \{v_1, v_2/f, x\}e_2$.

By inverting the typing of e, we have
$v_1 : \tau_2 \rightarrow \tau$ and $v_2 : \tau_2$.

By inverting the typing of v_1, we have $[f{:}\tau_2 \rightarrow \tau][x{:}\tau_2] \vdash e_2 : \tau$.

By substitution we have $\{v_1, v_2/f, x\}e_2 : \tau$, as required.

Proof of Preservation for Search Rules

We have $e = +(e_1, e_2)$, $e' = +(e_1', e_2)$, and $e_1 \mapsto e_1'$.

By inversion $e_1 : \texttt{int}$, so that by induction $e_1' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

Proof of Preservation for Search Rules

We have $e = \texttt{+}(v_1, e_2)$, $e' = \texttt{+}(v_1, e_2')$, and $e_2 \mapsto e_2'$.

By inversion $e_2 : \texttt{int}$, so that by induction $e_2' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

Proof of Preservation for Search Rules

We have $e = \texttt{if}_\tau\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi}$ and $e' = \texttt{if}_\tau\, e_1' \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi}$.

By inversion we have that $e_1 : \texttt{bool}$, $e_2 : \tau$ and $e_3 : \tau$.

By inductive hypothesis $e_1' : \texttt{bool}$, and hence $e' : \tau$.

Proof of Preservation for Search Rules

There are two cases for application.

First, we have $e = \texttt{apply}(e_1, e_2)$
and $e' = \texttt{apply}(e'_1, e_2)$.

By inversion $e_1 : \tau_2 \rightarrow \tau$ and $e_2 : \tau_2$, for some type τ_2.

By induction $e'_1 : \tau_2 \rightarrow \tau$, and hence $e' : \tau$.

Proof of Preservation for Search Rules

Second, we have $e = \texttt{apply}(v_1, e_2)$ and $e' = \texttt{apply}(v_1, e_2')$.

By inversion, $v_1 : \tau_2 \rightarrow \tau$ and $e_2 : \tau_2$, for some type τ_2.

By induction $e_2' : \tau_2$, and hence $e' : \tau$.

Proof of Preservation

This completes the proof. How might it have failed?

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

$$\mathbf{=}(m,n) \mapsto \begin{cases} 1 & \textit{if } m = n \\ 0 & \textit{if } m \neq n \end{cases}$$

Then preservation would **fail**.

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

Proof of Preservation

Notice that if an instruction is **undefined**, this does not disturb preservation!

For example, if we **omitted** the instruction for =(m, n), the proof would still go through!

In other words, **preservation alone is not enough to characterize safety**.

Canonical Forms Lemma

The type system predicts the forms of values:

Lemma 2 (Canonical Forms)
Suppose that $v : \tau$ *and* v value*.*

1. *If* $\tau = \texttt{bool}$*, then either* $v = \texttt{true}$ *or* $v = \texttt{false}$*.*

2. *If* $\tau = \texttt{int}$*, then* $v = n$ *for some* n*.*

3. *If* $\tau = \tau_1 \rightarrow \tau_2$*, then* $v = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2 = e$ *for some* f*,* x*, and* e*.*

Proof of Canonical Forms Lemma

The proof is by **induction on typing**. For example, for $v : \texttt{bool}$,

- v cannot be a numeral, because $\texttt{int} \neq \texttt{bool}$.
- v cannot be a variable, because it is closed.
- v can be a boolean constant, as specified.
- v cannot be an application of a primitive, nor a function, nor an application of a function.

Proof of Progress

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? How do we proceed?

$\blacksquare$

Proof of Progress

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

For numerals, boolean constants, or functions, the result is immediate because they are values.

Consider the rule for typing addition expressions. We have $e = \texttt{+}(e_1, e_2)$ and $\tau = \texttt{int}$, with $e_1 : \texttt{int}$ and $e_2 : \texttt{int}$.

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

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

If e_1 is a value, then we note that by the canonical forms lemma $e_1 = n_1$ for some n_1, and we consider e_2.

By induction either e_2 is a value, or $e_2 \mapsto e_2'$ for some expression e_2'.

If e_2 is a value, then by the canonical forms lemma $e_2 = n_2$ for some n_2, and we note that $e \mapsto e'$, where $e' = n_1 + n_2$.

If e_2 is not a value, then $e \mapsto e'$, where $e' = \text{+}(v_1, e_2')$.

Proof of Progress

Suppose that $e = \texttt{apply}(e_1, e_2)$, with $e_1 : \tau_2 \to \tau$ and $e_2 : \tau_2$.

By the first 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 $e \mapsto \texttt{apply}(e_1', e_2)$ by the rule for evaluating applications, as required.

Proof of Progress

By the second inductive hypothesis, either e_2 is a value, or there exists e_2' such that $e_2 \mapsto e_2'$.

If e_2 is not a value, then $e \mapsto e'$, where $e' = \texttt{apply}(e_1, e_2')$, as required.

Finally, if both e_1 and e_2 are values, then by the Canonical Forms Lemma,

$$e_1 = \mathtt{fun}\, f\ (x{:}\tau_2){:}\tau\ =\ e''$$

and $e \mapsto e'$, where $e' = \{e_1, e_2/f, x\}e''$, by the rule for executing applications.

Proof of Progress

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

1. Some instruction step was omitted. If there were no instructions for =(n_1, n_2), then progress would fail.

2. Some search rule was omitted. If there were no rule for, say, =(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 embarassing situation!

Extending the Language

We deliberately omitted division from the language. Suppose we add `div` as a primitive operation and define the following evaluation rules for it:

$$\frac{(n_2 \neq 0)}{\texttt{div}(n_1, n_2) \mapsto n_1 \div n_2}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{div}(e_1, e_2) \mapsto \texttt{div}(e_1', e_2)}$$

$$\frac{e_1 \ \textsf{value} \qquad e_2 \mapsto e_2'}{\texttt{div}(e_1, e_2) \mapsto \texttt{div}(e_1, e_2')}$$

Extending the Language

Suppose the static semantics gives the following typing to `div`:

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash \mathtt{div}(e_1, e_2)}$$

Is the language still safe?

- Preservation continues to hold: new instruction preserves type.
- **Progress fails**: $\mathtt{div}(10, 0) \not\mapsto$, yet has type `int`.

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 **nzint** of non-zero integers. Revise the typing rule for division to:

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{nzint}}{\Gamma \vdash \mathtt{div}(e_1, e_2) : \mathtt{int}}$$

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 $e : \mathtt{int}$ evaluates to a non-zero integer.

Modifying the Dynamic Semantics

Idea: introduce a well-defined **error** transitions corresponding to **checked errors** such as zero denominator or array index out of bounds.

- Undefined transitions correspond to "core dumps". Eliminate them by giving them a well-defined meaning, namely error.

- Revise statement of safety to account for errors. A program has an **answer** that is either a value or an error.

Adding Errors

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.

Adding Errors

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

$$\frac{}{\texttt{div}(m, 0) \mapsto \texttt{error}}$$

Then we must propagate errors upwards:

$$\frac{}{\texttt{div}(\texttt{error}, e) \mapsto \texttt{error}}$$

$$\frac{v \ \mathsf{value}}{\texttt{div}(v, \texttt{error}) \mapsto \texttt{error}}$$

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

Adding Errors

Revise preservation and progress:

- **Preservation**: if $e : \tau$ and $e \mapsto e'$, where $e' \neq \texttt{error}$, then $e' : \tau$.

- **Progress**: if $e : \tau$, then either e is a value or e is `error` or there exists e' such that $e \mapsto e'$.

The proofs are largely the same. There must be "enough" propagation rules for progress to hold.

Summary

- Type safety expresses the **coherence** of the static and dynamic semantics.

- Coherence is elegantly expressed as the conjunction of **preservation** and **progress**.

Summary

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.