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?

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

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.

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.

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

Therein lies the art of type system design.

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 \( e \rightarrow e' \), then \( e' : \tau \).

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

Consequently, if \( e : \tau \) and \( e \rightarrow^* v \), then \( v : \tau \).
Formalization of Type Safety

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

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

Thus if \( e : \text{int} \) and \( e \rightsquigarrow v \), then \( v = n \) for some \( n \). In words: expressions of type int evaluate to numbers.

Proof of Preservation for Instruction Steps

The primitive operations are straightforward:

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

Clearly \( e' : \text{int} \), as required.

The other primitive operations are handled similarly.

Proof of Preservation for Instruction Steps

There are two cases for conditionals:

1. We have \( e = \text{if } \text{true} \text{ then } e_1 \text{ else } e_2 \) fi and \( e' = e_1 \).
   
   Since \( e : \tau \), we have \( e_1 : \tau \), by inversion.

2. We have \( e = \text{if } \text{false} \text{ then } e_1 \text{ else } e_2 \) fi and \( e' = e_2 \).
   
   Since \( e : \tau \), we have \( e_2 : \tau \), by inversion.

Proof of Preservation for Search Rules

We have \( e = +(e_1, e_2) \), \( e' = +(e'_1, e_2) \), and \( e_1 \rightsquigarrow e'_1 \).

By inversion \( e_1 : \text{int} \), so that by induction \( e'_1 : \text{int} \), and hence \( e' : \text{int} \), as required.
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

\[ \pi(m, n) \mapsto \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases} \]

Then preservation would fail.

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

Proof of Preservation for Search Rules
We have \( e = \langle v_1, c_2 \rangle \), \( e' = \langle v_1, c_2' \rangle \), and \( c_2 \mapsto c_2' \). By inversion \( c_2 : \text{int} \), so that by induction \( c_2' : \text{int} \), and hence \( e' : \text{int} \), as required.

Proof of Preservation for Search Rules
There are two cases for application.

First, we have \( e = \text{apply}(e_1, c_2) \) and \( e' = \text{apply}(e_1', c_2) \). By inversion \( e_1 : \tau_2 \rightarrow \tau \) and \( c_2 : \tau_2 \), for some type \( \tau_2 \). By induction \( e_1' : \tau_2 \rightarrow \tau \), and hence \( e' : \tau \).

Proof of Preservation for Search Rules
Second, we have \( e = \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 \) and \( e' = \text{if}\, e_1' \,\text{then}\, e_2 \,\text{else}\, e_3 \). By inductive hypothesis \( e_1' : \text{bool} \), and hence \( e' : \tau \).

Proof of Preservation
Notice that if an instruction is undefined, this does not disturb preservation!

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

In other words, preservation alone is not enough to characterize safety.
Proof of Progress

Theorem 3 (Progress)
If \( e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e \rightarrow e' \).

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 Canonical Forms Lemma

The proof is by induction on typing. For example, for \( v : \text{bool} \),
- \( v \) cannot be a numeral, because \( \text{int} \neq \text{bool} \).
- \( v \) 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.

Canonical Forms Lemma

The type system predicts the forms of values:

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

1. If \( \tau = \text{bool} \), then either \( v = \text{true} \) or \( v = \text{false} \).

2. If \( \tau = \text{int} \), then \( v = n \) for some \( n \).

3. If \( \tau = \tau_1 \rightarrow \tau_2 \), then \( v = \text{fun} (x:\tau_1) : \tau_2 \) is closed for some \( x, \) and \( e \).

Proof of Progress

If \( e_1 \rightarrow e'_1 \), then \( e \rightarrow e' \), where \( e' = + (e'_1, e_2) \), 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 \( e_2 \rightarrow e'_2 \) 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 \( e \rightarrow e' \), where \( e' = n_1 + n_2 \).

If \( e_2 \) is not a value, then \( e \rightarrow e' \), where \( e' = + (n_1, e'_2) \).

Proof of Progress

Suppose that \( e = \text{apply}(e_1, e_2) \), with \( e_1 : \tau_2 \rightarrow \tau \) and \( e_2 : \tau_2 \).

By the first inductive hypothesis, either \( e_1 \) is a value, or there exists \( e'_1 \) such that \( e_1 \rightarrow e'_1 \).

If \( e_1 \) is not a value, then \( e \rightarrow \text{apply}(e'_1, e_2) \) 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 $e_2 \rightarrow e'_2$.

If $e_2$ is not a value, then $e \rightarrow e'$, where $e' = \text{apply}(e_1, e'_2)$, as required.

Finally, if both $e_1$ and $e_2$ are values, then by the Canonical Forms Lemma, $e_1 = \text{fun}\{x,y\}:\tau . \text{is}\{e''\}$ end, and $e \rightarrow e'$, where $e' = \{e_1, e_2/f, x\}e''$, by the rule for executing applications.

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:

\[
\begin{align*}
\text{div}(n_1, n_2) &\rightarrow n_1 \div n_2 \\
\text{div}(e_1, e_2) &\rightarrow \text{div}(e'_1, e_2) \\
\text{if } e_1 \text{ value } &\rightarrow e'_2 \\
\text{div}(e_1, e_2) &\rightarrow \text{div}(e_1, e'_2)
\end{align*}
\]

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 $\eta(n_1, n_2)$, then progress would fail.

2. Some search rule was omitted. If there were no rule for, say, $\eta(e_1, e_2)$, 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 Type System

A natural idea: add a type \texttt{nzint} of non-zero integers. Revise the typing rule for division to:

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

But how do we “create” expressions of type \texttt{nzint}?

- This type does not have good closure properties, e.g. is not closed under subtraction.

- It is undecidable in general whether $e : \text{int}$ evaluates to a non-zero integer.

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.

Proof of Progress

zow could the proof have failed?

Proof of Progress

exists $w$ failed the support...
Modifying the Dynamic Semantics

Idea: introduce 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

For example, we add an error transition for zero divisor:
\[
\text{div}(m, 0) \rightarrow \text{error}
\]
Then we must propagate errors upwards:
\[
\begin{align*}
\text{div}(\text{error}, e) & \rightarrow \text{error} \\
\text{div}(e, \text{error}) & \rightarrow \text{error}
\end{align*}
\]
and so on for the other non-value expression forms.

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.

Revise preservation and progress:

- Preservation: if \(e : \tau\) and \(e \rightarrow e'\), where \(e' \neq \text{error}\), then \(e' : \tau\).
- Progress: if \(e : \tau\), then either \(e\) is a value or \(e\) is error or there exists \(e'\) such that \(e \rightarrow e'\).

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.

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.