

## 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?

1

## 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!

2

## Type Safety

What is a run-time type error?

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

3

## 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.

4

## 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.

5

## 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.

6

## 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.

7

## 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.

8

## 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  $e$  val, or there exists  $e'$  such that  $e \mapsto e'$ .

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

9

## Formalization of Type Safety

Moreover, the type of a (closed) value determines its form. For  $\mathcal{L}\{\text{num str}\}$ , if  $v : \tau$ , then

- If  $\tau = \text{num}$ , then  $v = \text{num}[n]$  for some  $n$ .
- If  $\tau = \text{str}$ , then  $v = \text{str}[s]$  for some  $s$ .

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

10

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

### Theorem 1 (Preservation)

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

**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.

■

11

### Proof of Preservation for $\mathcal{L}\{\text{num str}\}$ Instruction Steps

The primitive operations are straightforward:

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

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

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

12

### Proof of Preservation for $\mathcal{L}\{\text{num str}\}$ Instruction Steps

The let instruction is a bit more complex. We require both the inversion and the substitution lemmas.

We have  $e = \text{let}(e_1; x.e_2) : \tau$  and  $e' = [e_1/x]e_2$ .

By inverting the typing of  $e$ , we have  $e_1 : \tau_1$  for some  $\tau_1$  such that  $x : \tau_1 \vdash e_2 : \tau$ .

By substitution (see the first Structural Properties of Typing slide from the previous lecture) we have  $[e_1/x]e_2 : \tau$ , as required.

13

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

Consider the search rule:

$$\frac{e_1 \mapsto e'_1}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e'_1; e_2)}$$

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

By inversion  $e_1 : \text{num}$  and  $e_2 : \text{num}$ . By induction  $e'_1 : \text{num}$ , and hence  $e' : \text{num}$ , as required. The case for the similar rule for multiplication is handled similarly.

14

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

Consider the search rule:

$$\frac{e_1 \text{ val } e_2 \mapsto e'_2}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e'_2)}$$

We have  $e = \text{plus}(e_1; e_2)$ ,  $e' = \text{plus}(e_1; e'_2)$ , and  $e_2 \mapsto e'_2$ .

By inversion  $e_1 : \text{num}$  and  $e_2 : \text{num}$ , so that by induction  $e'_2 : \text{num}$ , and hence  $e' : \text{num}$ , as required. The case for the similar rule for multiplication is handled similarly.

15

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

Consider the search rule:

$$\frac{e_1 \mapsto e'_1}{\text{cat}(e_1; e_2) \mapsto \text{cat}(e'_1; e_2)}$$

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

By inversion  $e_1 : \text{str}$  and  $e_2 : \text{str}$ . By induction  $e'_1 : \text{str}$ , and hence  $e' : \text{str}$ , as required.

16

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

Consider the search rule:

$$\frac{e_1 \text{ val } e_2 \mapsto e'_2}{\text{cat}(e_1; e_2) \mapsto \text{cat}(e_1; e'_2)}$$

We have  $e = \text{cat}(e_1; e_2)$ ,  $e' = \text{cat}(e_1; e'_2)$ , and  $e_2 \mapsto e'_2$ .

By inversion  $e_1 : \text{str}$  and  $e_2 : \text{str}$ , so that by induction  $e'_2 : \text{str}$ , and hence  $e' : \text{str}$ , as required.

17

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

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

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

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

18

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

This completes the proof. How might it have failed?

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

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

Then preservation would **fail**.

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

19

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

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

For example, if we **omitted** the instruction for  $\text{plus}(\text{num}[0]; \text{num}[0])$ , the proof would still go through!

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

20

### Canonical Forms Lemma

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

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

Suppose that  $e : \tau$  and  $e$  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  $n$ .

21

### 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.

22

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

**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. ■

23

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

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.

24

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

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  $e_2 \mapsto e'_2$  for some expression  $e'_2$ .

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  $e' = \text{plus}(e_1; e'_2)$ .

25

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

Suppose that  $e = \text{let}(e_1; x.e_2)$ . 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  $e \mapsto \text{let}(e'_1; x.e_2)$  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  $x : \tau_1 \vdash e_2 : \tau$  and  $e \mapsto e'$ , where  $e' = [e_1/x]e_2$ , by the rule for executing let expressions.

26

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

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,  $\text{plus}(e_1; e_2)$ , 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!

27

### Extending the $\mathcal{L}\{\text{num str}\}$ Language

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

$$\frac{(n_2 \neq 0)}{\text{div}(\text{num}[n_1]; \text{num}[n_2]) \mapsto \text{num}[n_1 \div n_2]}$$

$$\frac{e_1 \mapsto e'_1}{\text{div}(e_1; e_2) \mapsto \text{div}(e'_1; e_2)}$$

$$\frac{e_1 \text{ val } \quad e_2 \mapsto e'_2}{\text{div}(e_1; e_2) \mapsto \text{div}(e_1; e'_2)}$$

28

### Extending the $\mathcal{L}\{\text{num str}\}$ Language

Suppose the static semantics gives the following typing to  $\text{div}$ :

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \text{div}(e_1; e_2) : \text{num}}$$

Is the language still safe?

- Preservation continues to hold: new instruction preserves type.
- **Progress fails:**  $\text{div}(\text{num}[10]; \text{num}[0]) \not\mapsto$ , yet has type  $\text{num}$ .

29

## 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.

30

## Extending the Type System

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

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{nzint}}{\Gamma \vdash \text{div}(e_1; e_2) : \text{num}}$$

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.

31

## 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.

32

## Adding Errors

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

$$\frac{e_1 \text{ val}}{\text{div}(\text{num}[e_1]; \text{num}[0]) \text{ err}}$$

Then we must propagate errors upwards:

$$\frac{e_1 \text{ err}}{\text{plus}(e_1; e_2) \text{ err}}$$
$$\frac{e_1 \text{ val} \quad e_2 \text{ err}}{\text{plus}(e_1; e_2) \text{ err}}$$

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

33

## 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  $e \mapsto e'$ .

**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. ■

34

## 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, `error`, 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.

35

### Adding Errors

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

$$\frac{}{\mathbf{error} : \tau}$$

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.

36

### Summary

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

38

### Adding Errors

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

$$\frac{e_1 \text{ val}}{\mathbf{div}(e_1; \mathbf{num}[0]) \mapsto \mathbf{error}}$$

Then we must propagate errors upwards:

$$\frac{}{\mathbf{plus}(\mathbf{error}; e_2) \mapsto \mathbf{error}}$$

$$\frac{e_1 \text{ val}}{\mathbf{plus}(e_1; \mathbf{error}) \mapsto \mathbf{error}}$$

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

Defining  $e \text{ err}$  to hold exactly when  $e = \mathbf{error}$ , the Progress With Error theorem still holds.

37

### 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.

39