Lecture Notes 3
Type Safety

Carlo Angiuli

B522: PL Foundations
January 27, 2025

In this lecture, we use the concepts introduced in the previous lecture to state and prove our first interesting theorem about a programming language: *type safety* (or *type soundness*) for a first-order language with booleans and numbers.

Type safety for first-order languages is covered in Chapters 4–6 of Harper [Har16], but we will consider a different language taken from Chapter 8 of Pierce [Pie02] because we're covering topics in a different order from the textbook.

1 A slightly more interesting language

Our running example in the previous lecture was a simple programming language of boolean expressions. This lecture we'll consider a language with both booleans and natural numbers.

Definition 3.1. The judgment e $\mathtt{tm}$ is defined by the following BNF grammar:

$$\begin{array}{lrl} \textit{Terms} & e ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}(e,e,e) \\ & \mid & \mathtt{zero} \mid \mathtt{suc}(e) \mid \mathtt{pred}(e) \mid \mathtt{zero?}(e) \end{array}$$

We will define the meaning of this language shortly, but some helpful remarks: We've removed `not` to shorten our proofs; it's definable in terms of `if`. `pred` is short for "predecessor," as in the opposite of "successor"; it subtracts one. `zero?` is a boolean test of whether its (numerical) input is the number `zero`.

2 Operational semantics (dynamics)

At the end of the previous lecture, we defined the meaning of programs using a binary *evaluation* judgment $e \Downarrow e'$ stating that the expression e evaluates to the expression e'. We will do two things differently this time:

- Rather than defining a binary judgment for the "full" evaluation of a term, we will define a binary judgment for taking *a single step of computation*.
- We will define a unary judgment expressing that a term is *finished computing*.

The evaluation judgment $e \Downarrow e'$ from previous lecture is sometimes called *natural semantics* or *big-step operational semantics*; it corresponds to defining an interpreter by structural recursion.

Today's judgment $e \longmapsto e'$ is often called *structural operational semantics* or *small-step operational semantics* (for obvious reasons). Small-step judgments are used more commonly than evaluation judgments; there's nothing wrong with evaluation judgments, but small-step judgments are often easier to reason about, especially when proving type safety.

Remark 3.2. In both approaches, note that the intermediate and final stages of a computation are drawn from the same exact collection of terms as our input language. This is a convenient simplifying assumption when it works, but it doesn't always. For example, your interpreters in C311/B521 evaluated `lambda`s to "closures," which were not part of the input language.

Definition 3.3 (Values)**.** For e tm, we define the judgment e val ("e is a value") by the following inference rules. We notate values using the metavariable v.

$$\frac{}{\mathtt{true}\ \mathsf{val}} \qquad \frac{}{\mathtt{false}\ \mathsf{val}} \qquad \frac{}{\mathtt{zero}\ \mathsf{val}} \qquad \frac{v\ \mathsf{val}}{\mathtt{suc}(v)\ \mathsf{val}}$$

Values are programs that are "finished computing."

Definition 3.4 (Small-step operational semantics)**.** For e tm, we define the judgment $e \longmapsto e'$ ("e steps to e'") by the inference rules in Figure 3.1.

The reader may notice that there are two distinct "kinds of rules" in small-step operational semantics. Rules like

$$\frac{}{\mathtt{if}(\mathtt{true}, e_2, e_3) \longmapsto e_2}$$

are sometimes called *principal reductions*; they define how the primitive operations behave on values. On the other hand, rules like

$$\frac{e_1 \longmapsto e_1'}{\mathtt{if}(e_1, e_2, e_3) \longmapsto \mathtt{if}(e_1', e_2, e_3)}$$

define the language's *evaluation order*; in this language, `if` fully evaluates its first argument but not its second or third arguments. (Harper [Har16] calls these rules "search transitions.") We say that the first argument of `if` is the *principal argument*.

$$\frac{}{\texttt{if}(\texttt{true}, e_2, e_3) \longmapsto e_2} \qquad \frac{}{\texttt{if}(\texttt{false}, e_2, e_3) \longmapsto e_3}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{if}(e_1, e_2, e_3) \longmapsto \texttt{if}(e_1', e_2, e_3)} \qquad \frac{e \longmapsto e'}{\mathsf{suc}(e) \longmapsto \mathsf{suc}(e')}$$

$$\frac{}{\mathsf{pred}(\mathsf{zero}) \longmapsto \mathsf{zero}} \star \qquad \frac{v \ \mathsf{val}}{\mathsf{pred}(\mathsf{suc}(v)) \longmapsto v} \qquad \frac{e \longmapsto e'}{\mathsf{pred}(e) \longmapsto \mathsf{pred}(e')}$$

$$\frac{}{\texttt{zero?}(\texttt{zero}) \longmapsto \texttt{true}} \qquad \frac{v \ \mathsf{val}}{\texttt{zero?}(\texttt{suc}(v)) \longmapsto \texttt{false}}$$

$$\frac{e \longmapsto e'}{\texttt{zero?}(e) \longmapsto \texttt{zero?}(e')}$$

Figure 3.1: Definition of $\longmapsto$.

Remark 3.5. There are also two kinds of term operators: the ones that "make data" and the ones that "consume data." `true`, `false`, `zero`, and `suc` make data, and are the values of our language. `if`, `pred`, and `zero?` consume data; each of them has a search transition rule and several principal reductions.

Remark 3.6. Figure 3.1 contains one possibly dubious rule: $\mathsf{pred}(\mathsf{zero}) \longmapsto \mathsf{zero}$. We will revisit this rule later.

The following are "sanity check" lemmas that we can prove by rule induction.

Lemma 3.7. *If v* val *then v* tm.

Lemma 3.8. *If e* tm *and $e \longmapsto e'$ then e'* tm.

Lemma 3.9 (Determinacy)**.** *If $e \longmapsto e'$ and $e \longmapsto e''$ then $e' = e''$.*

Lemma 3.10 (Finality of values)**.** *If v* val *then there is no e' such that $v \longmapsto e'$.*

Remark 3.11. You might be tempted to also state the following lemma:

"If e tm then either $e \longmapsto e'$ or e val."

Unfortunately, that is **false**: terms such as `zero?(true)` and $\texttt{if}(\texttt{zero}, e_2, e_3)$ are neither values nor take a step, because their principal arguments are somehow "the wrong kind of thing." Such terms are called *stuck*.

Definition 3.12. For e tm, we define the judgment $e \longmapsto^* e'$ ("e takes zero or more steps to e'") by the following inference rules:

$$\frac{}{e \longmapsto^* e} \qquad\qquad \frac{e \longmapsto e' \qquad e' \longmapsto^* e''}{e \longmapsto^* e''}$$

$\longmapsto^*$ is called the *reflexive transitive closure of* $\longmapsto$.

Lemma 3.13 (Uniqueness of values)**.** *If $e \longmapsto^* v$ and $e \longmapsto^* v'$ where v val and v' val, then $v = v'$.*

We typically don't write derivation trees for $\longmapsto$ or $\longmapsto^*$ because they are quite awkwardly shaped. Instead, we typically write a sequence of single-step reductions like so:

$$\begin{aligned}
&\mathtt{suc(if(zero?(}\underline{\mathtt{pred(suc(zero))}}\mathtt{), suc(zero), zero))}\\
\longmapsto\ &\mathtt{suc(if(}\underline{\mathtt{zero?(zero)}}\mathtt{, suc(zero), zero))}\\
\longmapsto\ &\mathtt{suc(}\underline{\mathtt{if(true, suc(zero), zero)}}\mathtt{)}\\
\longmapsto\ &\mathtt{suc(suc(zero))}
\end{aligned}$$

The underlines are optional but indicate the subterm that is being simplified in the following step. A term that matches the left-hand side of a principal reduction is called a *reducible expression*, or *redex* for short.

Remark 3.14. The plural of redex is redexes, **not** redices.

3 Type system (statics): 2 Types 2 Furious

Because our language has two different kinds of data in it, it is possible for evaluation to get stuck if an operator that expects a number is given a boolean, or vice versa. We consider programs that eventually get stuck to be erroneous or nonsense, and would like to exclude them from consideration *before we even try to evaluate them.* We do this by syntactically characterizing which programs produce numerical values, and which programs produce boolean values.

$$\textit{Types} \quad \tau ::= \quad \mathtt{bool} \mid \mathtt{num}$$

Definition 3.15 (Type system)**.** For e tm and τ ty, we define the judgment $e : \tau$

("e has type τ") by the following inference rules:

$$\frac{}{\texttt{true} : \texttt{bool}} \qquad \frac{}{\texttt{false} : \texttt{bool}} \qquad \frac{}{\texttt{zero} : \texttt{num}} \qquad \frac{e : \texttt{num}}{\texttt{suc}(e) : \texttt{num}}$$

$$\frac{e : \texttt{num}}{\texttt{pred}(e) : \texttt{num}} \qquad \frac{e : \texttt{num}}{\texttt{zero?}(e) : \texttt{bool}} \qquad \frac{e_1 : \texttt{bool} \qquad e_2 : \tau \qquad e_3 : \tau}{\texttt{if}(e_1, e_2, e_3) : \tau}$$

Exercise 3.16. Show there is no τ ty such that $\texttt{zero?(true)} : \tau$.

Lemma 3.17 (Uniqueness of types)**.** *If $e : \tau$ and $e : \tau'$ then $\tau = \tau'$.*

Note that our type system does not refer to our operational semantics. Rather, it is a purely *static* analysis of the syntactic structure of terms: $e : \texttt{num}$ never gets stuck, and its only possible values are natural numbers; $e : \texttt{bool}$ never gets stuck, and its only possible values are `true` and `false`. In the next section, we will show that our analysis is *sound* (Corollary 3.24), but our analysis (like essentially all type systems) is not *complete*. That is, it is a conservative analysis that "misses" some programs that do actually compute boolean or numerical values.

Exercise 3.18. Show that `if(true, true, zero)` does not have type `bool` (in fact it has no type at all), but it steps to `true`.

4 Type safety = progress + preservation

The type safety theorem, also known as type soundness, expresses that our type system and operational semantics agree with one another.

Theorem 3.19 (Type safety)**.**

1. *If $e : \tau$ and $e \longmapsto e'$ then $e' : \tau$.*
2. *If $e : \tau$ then either e val or $e \longmapsto e'$.*

The first clause, *preservation* (or *subject reduction*), says that $\longmapsto$ preserves typing. The second clause, *progress*, says that well-typed terms are never "stuck": they either are values or can take a step. Putting these together, "well-typed programs do not go wrong"!

Remark 3.20. Progress may sound like it alone implies that "well-typed programs don't get stuck," but it really says that well-typed programs aren't *immediately* stuck. We need preservation to conclude that well-typed programs never become stuck during evaluation, immediately or otherwise.

We prove type safety in three steps: canonical forms (Lemma 3.21), progress (Lemma 3.22), and preservation (Lemma 3.23).

Lemma 3.21 (Canonical forms)**.** *Suppose $v : \tau$ and v* val*. Then:*

1. *If $\tau = \texttt{bool}$, then $v = \texttt{true}$ or $v = \texttt{false}$.*
2. *If $\tau = \texttt{num}$, then v* nat *where the* nat *judgment is defined as:*

$$\frac{}{\texttt{zero}\ \mathsf{nat}} \qquad\qquad \frac{v\ \mathsf{nat}}{\texttt{suc}(v)\ \mathsf{nat}}$$

Proof. We use rule induction on $v : \tau$ to prove $P(v, \tau)$ = "If v val then (1) if $\tau = \texttt{bool}$ then $v = \texttt{true}$ or $v = \texttt{false}$, and (2) if $\tau = \texttt{num}$ then v nat." (The following proof is written out in extra detail, hopefully for clarity.)

- Case $\dfrac{}{\texttt{true} : \texttt{bool}}$:

  The antecedent $\texttt{true}$ val holds. For (1), the antecedent $\tau = \texttt{bool}$ holds, and $v = \texttt{true}$. For (2), the antecedent $\tau = \texttt{nat}$ is false so the statement holds vacuously.

- Case $\dfrac{}{\texttt{false} : \texttt{bool}}$:

  Analogous to previous case.

- Case $\dfrac{}{\texttt{zero} : \texttt{num}}$:

  $\texttt{zero}$ val holds. (1) is vacuous; for (2), $\tau = \texttt{num}$ holds, and $\texttt{zero}$ nat.

- Case $\dfrac{e : \texttt{num}}{\texttt{suc}(e) : \texttt{num}}$:

  Suppose $\texttt{suc}(e)$ val; then by inversion, we have e val. (1) is vacuous because $\tau \neq \texttt{bool}$; for (2), $\tau = \texttt{num}$ holds, and we must show $\texttt{suc}(e)$ nat holds. By our inductive hypothesis $P(e, \texttt{num})$, if e val and $\texttt{num} = \texttt{num}$ then e nat. Thus e nat and so $\texttt{suc}(e)$ nat as required.

- Remaining cases ($\texttt{pred}$, $\texttt{zero?}$, $\texttt{if}$): The antecedent v val is false by inversion, so the statement holds vacuously. □

Lemma 3.22 (Progress)**.** *If $e : \tau$ then either e* val *or $e \longmapsto e'$ for some e'.*

Proof. By rule induction on $e : \tau$.

- Case $\dfrac{}{\texttt{true} : \texttt{bool}}$:

  True by $\texttt{true}$ val.

- Case $\dfrac{}{\texttt{false} : \texttt{bool}}$:

  True by $\texttt{false}$ val.

- Case $\dfrac{}{\texttt{zero} : \texttt{num}}$:

  True by $\texttt{zero}$ val.

- Case $\dfrac{e : \texttt{num}}{\texttt{suc}(e) : \texttt{num}}$:

  We must show that $\texttt{suc}(e)$ val or $\texttt{suc}(e) \longmapsto e'$. By our inductive hypothesis, either e val or $e \longmapsto e''$. In the former case, $\texttt{suc}(e)$ val; in the latter case, $\texttt{suc}(e) \longmapsto \texttt{suc}(e')$.

- Case $\dfrac{e : \texttt{num}}{\texttt{pred}(e) : \texttt{num}}$:

  This is never a value, so we will have to show $\texttt{pred}(e) \longmapsto e'$. By our inductive hypothesis, either e val or $e \longmapsto e''$. In the latter case, $\texttt{pred}(e) \longmapsto \texttt{pred}(e'')$. In the former case, by Lemma 3.21 we have e nat, and complete the proof by inversion on e nat. If $e = \texttt{zero}$, then $\texttt{pred}(\texttt{zero}) \longmapsto \texttt{zero}$. Otherwise, if $e = \texttt{suc}(v)$ where v nat, then $\texttt{pred}(\texttt{suc}(v)) \longmapsto v$.

- Case $\dfrac{e : \texttt{num}}{\texttt{zero?}(e) : \texttt{bool}}$:

  Similar to previous case.

- Case $\dfrac{e_1 : \texttt{bool} \qquad e_2 : \tau \qquad e_3 : \tau}{\texttt{if}(e_1, e_2, e_3) : \tau}$:

  This is never a value, so we will have to show $\texttt{if}(e_1, e_2, e_3) \longmapsto e'$. By our inductive hypothesis on e_1, either e_1 val or $e_1 \longmapsto e_1'$. In the latter case, $\texttt{if}(e_1, e_2, e_3) \longmapsto \texttt{if}(e_1', e_2, e_3)$. In the former case, by Lemma 3.21 we have $e_1 = \texttt{true}$ or $e_1 = \texttt{false}$. If $e_1 = \texttt{true}$ then $\texttt{if}(e_1, e_2, e_3) \longmapsto e_2$; if $e_1 = \texttt{false}$ then $\texttt{if}(e_1, e_2, e_3) \longmapsto e_3$. □

Lemma 3.23 (Preservation). *If $e : \tau$ and $e \longmapsto e'$ then $e' : \tau$.*

Proof. By rule induction on $e \longmapsto e'$.

Duplicative cases omitted.

- Case $\dfrac{}{\mathtt{if}(\mathtt{true}, e_2, e_3) \longmapsto e_2}$:

  By inversion on the typing judgment, if $\mathtt{if}(\mathtt{true}, e_2, e_3) : \tau$ then $e_2 : \tau$ and $e_3 : \tau$. Thus in particular $e_2 : \tau$ as required.

- Case $\dfrac{e_1 \longmapsto e_1'}{\mathtt{if}(e_1, e_2, e_3) \longmapsto \mathtt{if}(e_1', e_2, e_3)}$:

  By inversion on the typing judgment, if $\mathtt{if}(e_1, e_2, e_3) : \tau$ then $e_1 : \mathtt{bool}$, $e_2 : \tau$, and $e_3 : \tau$. By our inductive hypothesis, $e_1' : \mathtt{bool}$; the result follows by the typing rule for $\mathtt{if}$.

- Case $\dfrac{v\ \mathsf{val}}{\mathtt{pred}(\mathtt{suc}(v)) \longmapsto v}$:

  By two inversions on typing, if $\mathtt{pred}(\mathtt{suc}(v)) : \mathtt{num}$ then $v : \mathtt{num}$. □

Corollary 3.24. *If $e : \tau$ then either $e\ \mathtt{val}$ (in which case τ dictates the form of e, by Lemma 3.21) or $e \longmapsto e'$ and $e' : \tau$ (in which case we can apply this corollary again).*

Remark 3.25. Type safety tells us that well-typed terms do not get stuck, but it allows for the possibility that a well-typed term will get into an infinite loop without ever reaching a value. For this particular language it is easy to establish that all programs (even ill-typed ones!) terminate—every $\longmapsto$ rule shrinks the size of the term—but in more interesting languages we will need to resort to more sophisticated techniques to prove termination.

5 Evaluation contexts

Add more words in this section.

We can rephrase the operational semantics in Figure 3.1 to group all the “search transitions” into a single rule. Also called *contextual dynamics*, *context-sensitive reduction semantics*, and *reduction semantics*.

$$\textit{Evaluation contexts} \quad \mathcal{E} ::= \quad \circ \mid \mathtt{if}(\mathcal{E}, e, e) \mid \mathtt{suc}(\mathcal{E}) \mid \mathtt{pred}(\mathcal{E}) \mid \mathtt{zero?}(\mathcal{E})$$

Definition 3.26. For any evaluation context $\mathcal{E}$ and e tm, we define the instantiation of $\mathcal{E}$ with e, written $\mathcal{E}\{e\}$, by structural recursion:

$$\begin{aligned}
\circ\{e\} &= e \\
\texttt{if}(\mathcal{E}, e_2, e_3)\{e\} &= \texttt{if}(\mathcal{E}\{e\}, e_2, e_3) \\
\mathsf{suc}(\mathcal{E})\{e\} &= \mathsf{suc}(\mathcal{E}\{e\}) \\
\mathsf{pred}(\mathcal{E})\{e\} &= \mathsf{pred}(\mathcal{E}\{e\}) \\
\mathsf{zero?}(\mathcal{E})\{e\} &= \mathsf{zero?}(\mathcal{E}\{e\})
\end{aligned}$$

Isolate the "principal reductions" $e \longmapsto_p e'$ and define $\longmapsto$ in one rule.

$$\frac{e \longmapsto_p e'}{\mathcal{E}\{e\} \longmapsto \mathcal{E}\{e'\}} \qquad \frac{}{\texttt{if}(\texttt{true}, e_2, e_3) \longmapsto_p e_2} \qquad \frac{}{\texttt{if}(\texttt{false}, e_2, e_3) \longmapsto_p e_3}$$

$$\frac{}{\mathsf{pred}(\mathsf{zero}) \longmapsto_p \mathsf{zero}}\,\star \qquad \frac{v\ \mathsf{val}}{\mathsf{pred}(\mathsf{suc}(v)) \longmapsto_p v}$$

$$\frac{}{\mathsf{zero?}(\mathsf{zero}) \longmapsto_p \mathsf{true}} \qquad \frac{v\ \mathsf{val}}{\mathsf{zero?}(\mathsf{suc}(v)) \longmapsto_p \mathsf{false}}$$

6 Well-typed programs *can* go wrong

"Type errors" like zero?(true) are a major source of going wrong in a programming language, but sometimes things can go wrong even with the guardrails of a type system. Consider division by zero: there's no number for division by zero to step to, but statically ruling it out would require a type system that can detect which numerical expressions cannot be zero at runtime.

This may suggest that type safety does not hold for languages with division, but in fact we can refine the statement of type safety to account for this. The idea is to add a second kind of "outcome" to our language: in addition to evaluating to a value, a program may evaluate to a well-defined *error* state. This is distinct from a program failing to have a next state. We can think of the well-defined error states as *checked errors*, or errors that a user is forewarned might occur, and stuck terms as encountering *unchecked errors*, ones that indicate a gap in the language itself.

To see an example of this in action, let's introduce a new judgment e err indicating that e evaluates no further because it has encountered the "subtraction from zero" error. We can adjust the contextual dynamics of the previous section by deleting the principal reduction labeled $\star$ and replacing it with two new rules:

$$\frac{}{\texttt{pred(zero)} \longmapsto_p \texttt{zero}}\,\star \qquad \rightsquigarrow \qquad \frac{}{\texttt{pred(zero)}\ \mathsf{err}} \qquad \frac{e\ \mathsf{err}}{\mathcal{E}\{e\}\ \mathsf{err}}$$

The first rule says that `pred(zero)` raises the “subtraction from zero” error, and the second rule propagates this error through evaluation contexts.

The statements and proofs of the canonical forms and preservation lemmas remain unchanged, but we must adjust progress to account for this error:

Lemma 3.27 (Progress)**.** *If $e : \tau$ then either e val or e err or $e \longmapsto e'$ for some e'.*

Type safety then still tells us that programs never get stuck: the possible outcomes are now that a program terminates successfully with a value, or terminates unsuccessfully with the “subtraction from zero” error, or gets into an infinite loop.[4]

We will study this further in the next problem set.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages.* Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002. ISBN: 0-262-16209-1.

[4]Again, there are no infinite loops in this language, although type safety does not tell us this.