Welcome and Introduction
(and Crash-lecture on Fundamentals)

Matthew Fluet

Toyota Technological Institute at Chicago

Logic and Theorem Proving in Programming Languages
Oregon Programming Languages Summer School
July 22, 2008

School Overview and Themes

Lecture Topics and Speakers

SMT Solvers	Leonardo de Moura
Mechanization of Metatheory using LF and Twelf	Robert Harper
Compiler Construction in Formal Logical Frameworks	Jason Hickey
Specification and Verification of Programs with Pointers	Rustan Leino
Leveraging Domain-Specific Languages for Reasoning	Sorin Lerner
Reasoning About Programs with ACL2	Pete Manolios
Putting the Curry-Howard Isomorphism to Work	Tim Sheard
Nominal Techniques	Christian Urban
Coq for Programming Language Metatheory	Stephanie Weirich

Current research focused on integrating expressive logical systems and powerful theorem-proving assistants into the design, definition, implementation, and verification of programming languages and programs.

The Big Picture

Software is now a critical component of daily infrastructure.

- computers, airplanes, mass-transit systems, power grids

Important to be able to *formally* reason about the behavior of software.
Want to be able to make powerful "For all..." statements:

- For all executions of the air-traffic-control program, ...

Integration of logics and theorem-provers with programming languages is a crucial step towards wide deployment of verified software components.

The Small Picture

We won't be formally verifying air-traffic-control programs this summer.

We will see many core ideas – specification logics, automated and interactive theorem proving, type systems, etc.

A variety of techniques and tools for reasoning at different granularities:

- at the level of individual programs
- at the level of classes of related programs
- at the level of programming-language definitions

Rest of this lecture...

Some basic fundamentals:

- techniques for giving precise definitions of programming languages
  - without precise definitions, we can't reason formally about programs
- a technique for proving properties about all programs in a language
  - e.g., certain kinds of errors cannot happen in any program

Rest of this lecture...

Some basic fundamentals:

- Inductive definitions
  - basis for defining all kinds of logics, languages, and systems
- MinML – a little programming language
  - Syntax
  - Type system
  - Operational semantics
  - Type safety – a canonical PL proof

Acknowledgement: Many slides borrowed from a similar crash-lectures given by David Walker and Dan Grossman at previous OPLSS.

Reading and Study

- Benjamin Pierce, *Types and Programming Languages*
  - available at bookstores
- Bob Harper, *Practical Foundations for Programming Languages*
  - available online (`http://www.cs.cmu.edu/~rwh/plbook/book.pdf`)

Inductive Definitions

Inductive Definitions

An *inductive definition* consists of:

- One or more *judgements* (i.e., assertions)
- A set of *inference rules* for deriving judgements

Example:

- Judgement is "n **nat**"
- Inference rules are
  - "`zero` **nat**"
  - "if n **nat**, then `succ`(n) **nat**"

Inference Rule Notation

Inference rules are normally written as:

$$\frac{J_1 \qquad \cdots \qquad J_n}{J}$$

where J and $J_1, \ldots, J_n$ are judgements.

$J_1, \ldots, J_n$ are the *premises*.
J is the *conclusion*.

An inference rule with no premises is an *axiom*.
An inference rule with some premises is a *proper rule*.

Inference Rule Notation

Example: the inference rules for deriving n **nat** are:

$$\frac{}{\texttt{zero}\ \mathbf{nat}}\ \textsc{zero} \qquad \frac{n\ \mathbf{nat}}{\texttt{succ}(n)\ \mathbf{nat}}\ \textsc{succ}$$

The second rule is a *rule schema*.
It denotes an infinite collection of inference rules,
each one obtained by replacing the *parameter* n with a concrete object.

Also, optionally label (ZERO,SUCC) rules for easy reference.

Derivation of Judgements

A judgement J is *derivable* iff either

- there is an axiom

$$\frac{}{\mathsf{J}}$$

- or, there is a proper rule

$$\frac{\mathsf{J}_1 \qquad \cdots \qquad \mathsf{J}_n}{\mathsf{J}}$$

  such that $\mathsf{J}_1, \ldots, \mathsf{J}_n$ are derivable.

Derivation of Judgements

Determine whether a judgement is derivable by working backwards.

Example: the judgement $\texttt{succ(succ(zero))}\ \mathbf{nat}$ is derivable:

$$\dfrac{\dfrac{\dfrac{}{\texttt{zero}\ \mathbf{nat}}\ \textsc{zero}}{\texttt{succ(zero)}\ \mathbf{nat}}\ \textsc{succ}}{\texttt{succ(succ(zero))}\ \mathbf{nat}}\ \textsc{succ}$$

Binary Trees

A judgement t **tree** asserting that t is a binary tree, with rules:

$$\frac{}{\texttt{empty}\ \textbf{tree}}\ \textsc{empty} \qquad \frac{t_1\ \textbf{tree} \qquad t_2\ \textbf{tree}}{\texttt{node}(t_1, t_2)\ \textbf{tree}}\ \textsc{node}$$

The judgement $\texttt{node}(\texttt{empty}, \texttt{node}(\texttt{empty}, \texttt{empty}))$ **tree** is derivable:

$$\dfrac{\dfrac{}{\texttt{empty}\ \textbf{tree}} \qquad \dfrac{\dfrac{}{\texttt{empty}\ \textbf{tree}} \qquad \dfrac{}{\texttt{empty}\ \textbf{tree}}}{\texttt{node}(\texttt{empty}, \texttt{empty})\ \textbf{tree}}}{\texttt{node}(\texttt{empty}, \texttt{node}(\texttt{empty}, \texttt{empty}))\ \textbf{tree}}$$

Rule Induction

By defininition, every derivable judgement

- is the consequence of some rule,
- whose premises are derivable

Thus, the rules are an exhaustive description of the derivable judgements.

If we want to prove some property about a (derivable) judgement,
then we simply consider all of the rules that may appear in the derivation.

Rule Induction

To show that every derivable judgement has a property $\mathfrak{P}$, it is enough to show that

- for every rule

$$\frac{\mathsf{J}_1 \qquad \cdots \qquad \mathsf{J}_n}{\mathsf{J}}$$

  if $\mathsf{J}_1, \ldots, \mathsf{J}_n$ have the property $\mathfrak{P}$, then J has property $\mathfrak{P}$.

This is the principal of *rule induction*.

Rule Induction – n **nat**

To prove that "If n **nat** (is derivable), then property $\mathfrak{P}(n)$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of:

- $\mathfrak{P}(\texttt{zero})$ (corresponding to $\dfrac{}{\texttt{zero}\ \mathbf{nat}}\ \textsc{zero}$)
- if $\mathfrak{P}(n)$, then $\mathfrak{P}(\texttt{succ}(n))$ (corresponding to $\dfrac{n\ \mathbf{nat}}{\texttt{succ}(n)\ \mathbf{nat}}\ \textsc{succ}$)

This is just ordinary mathematical induction.

Rule Induction – t **tree**

To prove that "If t **tree** (is derivable), then property $\mathfrak{P}(t)$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of:

- $\mathfrak{P}(\texttt{empty})$ (corresponding to $\dfrac{}{\texttt{empty}\ \textbf{tree}}\ \text{EMPTY}$)
- if $\mathfrak{P}(t_1)$ and $\mathfrak{P}(t_2)$, then $\mathfrak{P}(\texttt{node}(t_1, t_2))$

  (corresponding to $\dfrac{t_1\ \textbf{tree} \qquad t_2\ \textbf{tree}}{\texttt{node}(t_1, t_2)\ \textbf{tree}}\ \text{NODE}$)

This is structural induction.

Inductive Definitions
Inductive Definitions in PL

Inductive Definitions in PL

In the formal study of programming languages,

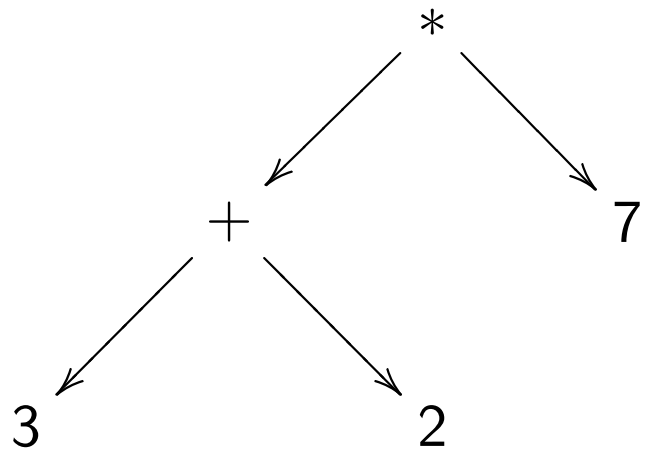
- we use inductive definitions to specify
  - abstract syntax
  - static semantics (typing)
  - dynamic semantics (evaluation)
  - other properties of programs and programming languages
- we use rule induction to prove
  - nearly everything

Concrete vs. Abstract Syntax

The *concrete syntax* of a program is a string of characters:

"(" "3" "+" "2" ")" "*" "7"

The *abstract syntax* of a program is a tree representing the structure and the computationally relevant portion of the program:



Concrete vs. Abstract Syntax

The concrete syntax contains many elements necessary for parsing:

- parentheses
- delimiters for comments
- rules for precedence of operators

The abstract syntax is much simpler;
grouping and precedence given directly by tree structure.

We work with simple abstract syntax, rather than complex concrete syntax.

Arithmetic Expressions, Informally

We consider a little language of arithmetic expressions.

Informally, an arithmetic exprssion *a* is

- a boolean value
- an `if` statement, with a test expression, a `then`-case expression, and an `else`-case expression
- the number zero
- the successor of an expression
- the predecessor of an expression
- a zero test of an expression

Arithmetic Expressions, Formally

The arithmetic expressions are defined by the judgement a **aexp**.

- a boolean value

$$\frac{}{\texttt{true}\ \textbf{aexp}}\ \textsc{True} \qquad \frac{}{\texttt{false}\ \textbf{aexp}}\ \textsc{False}$$

- an `if` statement, with a test expression, a `then`-case expression, and an `else`-case expression

$$\frac{a_b\ \textbf{aexp} \qquad a_t\ \textbf{aexp} \qquad a_f\ \textbf{aexp}}{\texttt{if}\ a_b\ \texttt{then}\ a_t\ \texttt{else}\ a_f\ \textbf{aexp}}\ \textsc{If}$$

- the number zero; the successor of an expression; the predecessor of an expression; and a zero test of an expression

$$\frac{}{\texttt{zero}\ \textbf{aexp}} \qquad \frac{a\ \textbf{aexp}}{\texttt{succ}(a)\ \textbf{aexp}} \qquad \frac{a\ \textbf{aexp}}{\texttt{pred}(a)\ \textbf{aexp}} \qquad \frac{a\ \textbf{aexp}}{\texttt{isZero}(a)\ \textbf{aexp}}$$

BNF

Defining every bit of syntax by inductive definitions is lengthy and tedious.

Syntax definitions are an especially simple form of inductive definition:

- context insensitive
- unary predicates

We use a convenient abbreviation: *BNF* (*Backus-Naur form*)

Arithmetic Expressions, BNF

The arithmetic expressions are inductively defined by these rules:

$$\begin{array}{rcl} a & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{if}\, a_b\, \texttt{then}\, a_t\, \texttt{else}\, a_f \mid \\ & & \texttt{zero} \mid \texttt{succ}(a) \mid \texttt{pred}(a) \mid \texttt{isZero}(a) \end{array}$$

Implicitly defines the judgement a **aexp**,
with 7 inductive rules corresponding to the 7 alternatives;

a is now a *syntax metavariable*; in the following, any use of a means an object a such that a **aexp** is derivable.

Two Functions over Arith. Exps.

We could also define via judgements:

$$
\begin{aligned}
\mathit{consts}(\texttt{true}) &= \{\texttt{true}\} \\
\mathit{consts}(\texttt{false}) &= \{\texttt{false}\} \\
\mathit{consts}(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) &= \mathit{consts}(a_b) \cup \mathit{consts}(a_t) \cup \mathit{consts}(a_f) \\
\mathit{consts}(\texttt{zero}) &= \{\texttt{zero}\} \\
\mathit{consts}(\texttt{succ}(a)) &= \mathit{consts}(a) \\
\mathit{consts}(\texttt{pred}(a)) &= \mathit{consts}(a) \\
\mathit{consts}(\texttt{isZero}(a)) &= \mathit{consts}(a)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{size}(\texttt{true}) &= 1 \\
\mathit{size}(\texttt{false}) &= 1 \\
\mathit{size}(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) &= 1 + \mathit{size}(a_b) + \mathit{size}(a_t) + \mathit{size}(a_f) \\
\mathit{size}(\texttt{zero}) &= 1 \\
\mathit{size}(\texttt{succ}(a)) &= 1 + \mathit{size}(a) \\
\mathit{size}(\texttt{pred}(a)) &= 1 + \mathit{size}(a) \\
\mathit{size}(\texttt{isZero}(a)) &= 1 + \mathit{size}(a)
\end{aligned}
$$

A Lemma and Proof

Lemma:
The number of distinct constants in any expression a is no greater than the size of a.

$$|consts(a)| \leq size(a)$$

Proof?

- By rule induction on the rules for the judgement a **aexp**
- Lingo: By induction on the structure of a

A Lemma and Proof

Lemma:
The number of distinct constants in any expression a is no greater than the size of a.

$$|consts(a)| \leq size(a)$$

Proof?

- By rule induction on the rules for the judgement a **aexp**
- Lingo: By induction on the structure of a

Structural Induction on Arith. Exps.

To prove that "If a **aexp** (is derivable), then property $\mathfrak{P}(a)$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of:

- True: $\mathfrak{P}(\texttt{true})$
- False: $\mathfrak{P}(\texttt{false})$
- If: if $\mathfrak{P}(a_b)$ and $\mathfrak{P}(a_t)$ and $\mathfrak{P}(a_f)$, then $\mathfrak{P}(\texttt{if}\ a_b\ \texttt{then}\ a_t\ \texttt{else}\ a_f)$
- Zero: $\mathfrak{P}(\texttt{zero})$
- Succ: if $\mathfrak{P}(a)$, then $\mathfrak{P}(\texttt{succ}(a))$
- Pred: if $\mathfrak{P}(a)$, then $\mathfrak{P}(\texttt{pred}(a))$
- IsZero: if $\mathfrak{P}(a)$, then $\mathfrak{P}(\texttt{isZero}(a))$

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

Proof:
By induction on the structure of a
(with $\mathfrak{P}(a) \equiv |consts(a)| \leq size(a)$).

- True, False: $\cdots$
- If: $\cdots$
- Zero: $\cdots$
- Succ, Pred, IsZero: $\cdots$

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

Proof:
By induction on the structure of a
(with $\mathfrak{P}(a) \equiv |consts(a)| \leq size(a)$).

- True:
  Show $|consts(\texttt{true})| \leq size(\texttt{true})$.

$$\begin{aligned} |consts(\texttt{true})| &= |\{\texttt{true}\}| && \text{(defn. of } consts\text{)} \\ &= 1 && \text{(property of } |\cdot|\text{)} \\ &= size(\texttt{true}) && \text{(defn. of } size\text{)} \end{aligned}$$

- False, Zero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

Proof:
By induction on the structure of a
(with $\mathfrak{P}(a) \equiv |consts(a)| \leq size(a)$).

- True:
  Show $|consts(\texttt{true})| \leq size(\texttt{true})$.

$$\begin{aligned} |consts(\texttt{true})| &= |\{\texttt{true}\}| && \text{(defn. of } consts\text{)} \\ &= 1 && \text{(property of } |\cdot|\text{)} \\ &= size(\texttt{true}) && \text{(defn. of } size\text{)} \end{aligned}$$

- False, Zero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

- If:
  Have $|consts(a_b)| \leq size(a_b)$ and $|consts(a_t)| \leq size(a_t)$
  and $|consts(a_f)| \leq size(a_f)$.
  Show $|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \leq size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)$.

$$
\begin{aligned}
&|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \\
&\quad = |consts(a_b) \cup consts(a_t) \cup consts(a_f)| && \text{(defn. of } consts\text{)} \\
&\quad \leq |consts(a_b)| + |consts(a_t)| + |consts(a_f)| && \text{(property of } |\cdot| \text{ and } \cup\text{)} \\
&\quad \leq size(a_b) + size(a_t) + size(a_f) && \text{(by induction hypothesis)} \\
&\quad < 1 + size(a_b) + size(a_t) + size(a_f) && \text{(property of } +\text{)} \\
&\quad = size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) && \text{(defn. of } size\text{)}
\end{aligned}
$$

- Succ, Pred, IsZero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

- If:
  Have $|consts(a_b)| \leq size(a_b)$ and $|consts(a_t)| \leq size(a_t)$
  and $|consts(a_f)| \leq size(a_f)$.
  Show $|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \leq size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)$.

  $$\begin{aligned}
  &|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \\
  &\quad = |consts(a_b) \cup consts(a_t) \cup consts(a_f)| && \text{(defn. of } consts\text{)} \\
  &\quad \leq |consts(a_b)| + |consts(a_t)| + |consts(a_f)| && \text{(property of } |\cdot| \text{ and } \cup\text{)} \\
  &\quad \leq size(a_b) + size(a_t) + size(a_f) && \text{(by induction hypothesis)} \\
  &\quad < 1 + size(a_b) + size(a_t) + size(a_f) && \text{(property of } +\text{)} \\
  &\quad = size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) && \text{(defn. of } size\text{)}
  \end{aligned}$$

- Succ, Pred, IsZero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

- If:
  Have $|consts(a_b)| \leq size(a_b)$ and $|consts(a_t)| \leq size(a_t)$
  and $|consts(a_f)| \leq size(a_f)$.
  Show $|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \leq size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)$.

  $$\begin{aligned}
  &|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \\
  &\quad = |consts(a_b) \cup consts(a_t) \cup consts(a_f)| && \text{(defn. of } consts\text{)} \\
  &\quad \leq |consts(a_b)| + |consts(a_t)| + |consts(a_f)| && \text{(property of } |\cdot| \text{ and } \cup\text{)} \\
  &\quad \leq size(a_b) + size(a_t) + size(a_f) && \text{(by induction hypothesis)} \\
  &\quad < 1 + size(a_b) + size(a_t) + size(a_f) && \text{(property of } +\text{)} \\
  &\quad = size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) && \text{(defn. of } size\text{)}
  \end{aligned}$$

- Succ, Pred, IsZero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

- If:
  Have $|consts(a_b)| \leq size(a_b)$ and $|consts(a_t)| \leq size(a_t)$
  and $|consts(a_f)| \leq size(a_f)$.
  Show $|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \leq size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)$.

  $$
  \begin{aligned}
  &|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \\
  &\quad = |consts(a_b) \cup consts(a_t) \cup consts(a_f)| && \text{(defn. of } consts\text{)} \\
  &\quad \leq |consts(a_b)| + |consts(a_t)| + |consts(a_f)| && \text{(property of } |\cdot| \text{ and } \cup\text{)} \\
  &\quad \leq size(a_b) + size(a_t) + size(a_f) && \text{(by induction hypothesis)} \\
  &\quad < 1 + size(a_b) + size(a_t) + size(a_f) && \text{(property of } +\text{)} \\
  &\quad = size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) && \text{(defn. of } size\text{)}
  \end{aligned}
  $$

- Succ, Pred, IsZero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

- If:
  Have $|consts(a_b)| \leq size(a_b)$ and $|consts(a_t)| \leq size(a_t)$
  and $|consts(a_f)| \leq size(a_f)$.
  Show $|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \leq size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)$.

  $$\begin{aligned}
  &|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \\
  &\quad = |consts(a_b) \cup consts(a_t) \cup consts(a_f)| && \text{(defn. of } consts\text{)} \\
  &\quad \leq |consts(a_b)| + |consts(a_t)| + |consts(a_f)| && \text{(property of } |\cdot| \text{ and } \cup\text{)} \\
  &\quad \leq size(a_b) + size(a_t) + size(a_f) && \text{(by induction hypothesis)} \\
  &\quad < 1 + size(a_b) + size(a_t) + size(a_f) && \text{(property of } +\text{)} \\
  &\quad = size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) && \text{(defn. of } size\text{)}
  \end{aligned}$$

- Succ, Pred, IsZero: Similar.

A Lemma and Proof

Lemma:
$|consts(a)| \leq size(a)$

- If:
  Have $|consts(a_b)| \leq size(a_b)$ and $|consts(a_t)| \leq size(a_t)$
  and $|consts(a_f)| \leq size(a_f)$.
  Show $|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \leq size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)$.

  $$
  \begin{aligned}
  &|consts(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f)| \\
  &\quad = |consts(a_b) \cup consts(a_t) \cup consts(a_f)| && \text{(defn. of } consts\text{)} \\
  &\quad \leq |consts(a_b)| + |consts(a_t)| + |consts(a_f)| && \text{(property of } |\cdot| \text{ and } \cup\text{)} \\
  &\quad \leq size(a_b) + size(a_t) + size(a_f) && \text{(by induction hypothesis)} \\
  &\quad < 1 + size(a_b) + size(a_t) + size(a_f) && \text{(property of } +\text{)} \\
  &\quad = size(\texttt{if } a_b \texttt{ then } a_t \texttt{ else } a_f) && \text{(defn. of } size\text{)}
  \end{aligned}
  $$

- Succ, Pred, IsZero: Similar.

What is a Proof?

A proof is an easily-checked justification of a proposition (e.g., a theorem).

Different people have different ideas about what "easily-checked" means.

The more formal a proof, the more "easily-checked".

- Formal enough, it can be checked by a computer.
- *Many*, *many* examples to come!

Next up: a canonical proof in PL – type safety.

MinML

MinML

Study MinML, a tiny fragment of ML

- integer and booleans
- recursive functions

Rich enough to be Turing complete,
but small enough to support a thorough analysis of its properties.

Defining MinML

What does it mean to define a programming language?

- Define the syntax of the programming language
  - what objects can be considered programs
- Define the semantics of the programming language
  - what does a program mean
  - how does a program execute (operational semantics)
  - what guarantees hold of program execution (static semantics)

MinML

Abstract Syntax of MinML

Abstract Syntax

The expressions of MinML are inductively defined by these rules:

$$\begin{array}{lrcl} \text{Numbers} & n & ::= & \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \\ \text{Expressions} & e & ::= & n \mid e_1 + e_2 \mid e_1 = e_2 \mid \cdots \mid \\ & & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e_b\, \texttt{then}\, e_t\, \texttt{else}\, e_f \mid \\ & & & x \mid \texttt{fun}\, f(x) \triangleright e \mid e_f\ e_a \end{array}$$

- x and f range over a set of *variables*

Bound and Free Variables

In the expression $\texttt{fun}\, f(x) \triangleright e$, the variables f and x are *bound*.

- Example: x is bound in $\texttt{fun}\, f(x) \triangleright x$.
- Example: x is bound in $\texttt{fun}\, f(x) \triangleright \texttt{fun}\, g(y) \triangleright \texttt{fun}\, h(z) \triangleright x\,(y\,z)$.
- Lingo: the *scope* of the variables f and x is the expression e.

Variables that are not bound are *free*.

- Example: x is free in $x\,y$.
- Example: x is free in $\texttt{fun}\, g(y) \triangleright x\,y$.

- Example: In $(\texttt{fun}\, f(x) \triangleright x)\,x$,
  the first occurrence of x is bound and
  the second occurrence of x is free.

An expression with no free variables is *closed*.

Variable Conventions

We use *standard conventions* involving bound variables:

- Expressions differing only in names of bound variables are equivalent:
  - Example: $(\texttt{fun}\, f(x) \triangleright x + 3) \equiv (\texttt{fun}\, g(y) \triangleright y + 3)$.
  - Lingo: we work with expressions "up to *alpha-conversion*".

- Can always *pick* bound variables to avoid clashes with other variables.
  - Lingo: we use the "*Barendregt convention*".

Substitution

The *capture-avoiding substitution* $e[e'/x]$
replaces all free occurrences of x with e' in e.

- Example: $(\texttt{fun}\, f(x) \triangleright x + y)[3/y] = (\texttt{fun}\, f(x) \triangleright x + 3)$.

Warning! Naïve substitution can "capture" free variables.
Rename bound variables during substitution to avoid "capture":

- Example: $(\texttt{fun}\, f(x) \triangleright x + y)[x/y] = (\texttt{fun}\, f(z) \triangleright z + x)$.

Bound/Free Variables; Variable Conventions; Substitution

I have not *formally* defined any of these notions.

Intended meanings are intuitively obvious,
but surprisingly tricky to define/use formally.

- Much more from Harper, Urban, and Weirich.

MinML

Static Semantics of MinML

Static Semantics

The *static semantics* (or *type system*)
imposes context-sensitive restrictions on programs.

- Distinguishes *well-typed* from *ill-typed* expressions.
- Well-typed programs (definitely) have well-defined behavior;
  ill-typed programs (may) have ill-defined behavior.
- In a well-typed program, every expression has a type.

Abstract Syntax for Types

The types of MinML are inductively defined by these rules:

$$\text{Types} \quad t \quad ::= \quad \texttt{num} \mid \texttt{bool} \mid t_d \rightarrow t_r$$

We redefine the expressions of MinML to annotate recursive functions with the types of their domain and range.

$$\text{Expressions} \quad e \quad ::= \quad \cdots \mid \texttt{fun}\, f(x : t_d) : t_r \triangleright e \mid \cdots$$

Typing Judgements

A typing judgement, or typing assertion, is a triple $\Gamma \vdash e : t$

- a type context Γ that assigns types to a set of variables,
- an expression e whose free variables are given by Γ, and
- a type t for the expression e in context Γ.

Read: "In context Γ, expression e has type t."

A type context is a finite function $\Gamma : \mathit{Vars} \longrightarrow \mathit{Types}$.
We write $\Gamma, x : t$ for the function Γ' defined as follows:

$$\begin{array}{rcll} \Gamma'(x) & = & t & \\ \Gamma'(y) & = & \Gamma(y) & \text{if } y \neq x \end{array}$$

Typing Rules

A variable has whatever type Γ assigns to it:

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textsc{Var}$$

The constants have the obvious types:

$$\frac{}{\Gamma \vdash n : \texttt{num}}\ \textsc{Num}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textsc{False}$$

Typing Rules

The numeric operations have the expected typing rules:

$$\frac{\Gamma \vdash e_1 : \mathtt{num} \qquad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash e_1 + e_2 : \mathtt{num}} \textsc{Add}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{num} \qquad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash e_1 = e_2 : \mathtt{bool}} \textsc{Eq}$$

Typing Rules

Conditionals:

$$\frac{\Gamma \vdash e_b : \texttt{bool} \qquad \Gamma \vdash e_t : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : t}\ \textsc{If}$$

- Both "branches" of a conditional must have the same type!
- Intuitively, we can't predict the outcome of the test, so we must insist that both results have the same type.
- Otherwise, we could not assign a unique type to the conditional.

Typing Rules

Applications:

$$\frac{\Gamma \vdash e_f : t_d \to t_r \qquad \Gamma \vdash e_a : t_d}{\Gamma \vdash e_f\ e_a : t_r}\ \textsc{App}$$

- Functions may only be applied to arguments in their domain, returning results in their range.

Typing Rules

Recursive functions:

$$\frac{\Gamma, f : t_d \to t_r, x : t_d \vdash e : t_r}{\Gamma \vdash \mathtt{fun}\, f(x : t_d) : t_r \triangleright e : t_d \to t_r}\ \textsc{Fun}$$

- We assume that $\{f, x\} \cap \mathrm{dom}(\Gamma) = \emptyset$.
  This is always possiby by our conventions on bound variables.
- The typing rule for a recursive function is tricky!
- We assume that
  - the function has the specified domain and range types, and
  - the argument has the specified domain type
- We verify that the body has the range type under these assumptions.
- If the assumptions are consistent,
  then the function really has the specified domain and range types.

Well-Typed and Ill-Typed Expressions

An expression e is *well-typed* in a context Γ
iff there exists a type t such that $\Gamma \vdash e : t$ (is derivable).

If there is no type t such that $\Gamma \vdash e : t$ (is derivable),
then expression e is *ill-typed* in context Γ.

Typing Example

Consider the following expression e_{fact}:

$$\begin{array}{l}\mathtt{fun}\, f\left(n : \mathtt{num}\right) : \mathtt{num} \triangleright \\ \qquad \mathtt{if}\, n = 0\, \mathtt{then}\, 1\, \mathtt{else}\, n * \left(f\left(n - 1\right)\right)\end{array}$$

Lemma: The expression e_{fact} has the type $\mathtt{num} \longrightarrow \mathtt{num}$.

To prove this, we must show that $\emptyset \vdash e_{\mathsf{fact}} : \mathtt{num} \longrightarrow \mathtt{num}$.

Typing Example

$$\mathfrak{D} = \dfrac{\dfrac{\Gamma(n) = \texttt{num}}{\Gamma \vdash n : \texttt{num}} \quad \dfrac{\dfrac{\Gamma(f) = \texttt{num} \to \texttt{num}}{\Gamma \vdash f : \texttt{num} \to \texttt{num}} \quad \dfrac{\dfrac{\Gamma(n) = \texttt{num}}{\Gamma \vdash n : \texttt{num}} \quad \dfrac{}{\Gamma \vdash 1 : \texttt{num}}}{\Gamma \vdash n - 1 : \texttt{num}}}{\Gamma \vdash f\,(n-1) : \texttt{num}}}{\Gamma \vdash n * (f\,(n-1)) : \texttt{num}}$$

$$\dfrac{\dfrac{\dfrac{\Gamma(n) = \texttt{num}}{\Gamma \vdash n : \texttt{num}} \quad \dfrac{}{\Gamma \vdash 0 : \texttt{num}}}{\Gamma \vdash n = 0 : \texttt{bool}} \quad \dfrac{}{\Gamma \vdash 1 : \texttt{num}} \quad \mathfrak{D} = \dfrac{\vdots}{\Gamma \vdash n * (f\,(n-1)) : \texttt{num}}}{\Gamma \vdash \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\,(n-1)) : \texttt{num}}$$

$$\dfrac{}{\emptyset \vdash \texttt{fun}\, f(n : \texttt{num}) : \texttt{num} \triangleright \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\,(n-1)) : \texttt{num} \to \texttt{num}}$$

- where $\Gamma = \emptyset, f : \texttt{num} \to \texttt{num}, n : \texttt{num}$

Typing Example

The typing rules tell us exactly when a program is well-typed and when it is ill-typed.

A *type checker* is a program that decides:

> *Given* Γ, e, *and* t,
> *does there exists a derivation* $\Gamma \vdash e : t$
> *according to the typing rules?*

Type Checking

How does the type checker find typing proofs?

Fact: the typing rules are *syntax directed*

- there is exactly one rule per expression form.

Therefore, the type checker can *invert* the typing rules
and work backwards from the expression to build the proof.

- Example: if the expression is a function,
  then the only possible proof applies the function typing rule.
- So, use that rule and check the body.

Summary of Static Semantics

The static semantics of MinML is specified by an inductive definition of the typing judgement $\Gamma \vdash e : t$.

$$\mathfrak{D} \quad = \quad \frac{\vdots}{\Gamma \vdash e : t}$$

Properties of the type system may be proved by induction (on typing derivations).

Induction on Typing Derivations

To prove that "If $\Gamma \vdash e : t$, then property $\mathfrak{P}(\Gamma, e, t)$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of:

- $\mathfrak{P}(\Gamma, n, \texttt{num})$
- if $\mathfrak{P}(\Gamma, e_1, \texttt{num})$ and $\mathfrak{P}(\Gamma, e_2, \texttt{num})$, then $\mathfrak{P}(\Gamma, e_1 + e_2, \texttt{num})$
- if $\mathfrak{P}(\Gamma, e_1, \texttt{num})$ and $\mathfrak{P}(\Gamma, e_2, \texttt{num})$, then $\mathfrak{P}(\Gamma, e_1 = e_2, \texttt{bool})$
- $\mathfrak{P}(\Gamma, \texttt{true}, \texttt{bool})$
- $\mathfrak{P}(\Gamma, \texttt{false}, \texttt{bool})$
- if $\mathfrak{P}(\Gamma, e_b, \texttt{bool})$ and $\mathfrak{P}(\Gamma, e_t, t)$ and $\mathfrak{P}(\Gamma, e_f, t)$,
  then $\mathfrak{P}(\Gamma, \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f, t)$
- if $\Gamma(x) = t$, then $\mathfrak{P}(\Gamma, x, t)$
- if $\mathfrak{P}(\Gamma, f : t_d \to t_r, x : t_d, e, t_r)$, then $\mathfrak{P}(\Gamma, \texttt{fun}\, f(x : t_d) : t_r \triangleright e, t_d \to t_r)$
- if $\mathfrak{P}(\Gamma, e_f, t_d \to t_r)$ and $\mathfrak{P}(\Gamma, e_a, t_d)$, then $\mathfrak{P}(\Gamma, e_f\ e_a, t_r)$

Properties of Typing

Lemma (Inversion):

- if $\Gamma \vdash n : t$, then $t = \texttt{num}$.
- if $\Gamma \vdash e_1 + e_2 : t$, then $\Gamma \vdash e_1 : \texttt{num}$ and $\Gamma \vdash e_2 : \texttt{num}$ and $t = \texttt{num}$.
- if $\Gamma \vdash e_1 = e_2 : t$, then $\Gamma \vdash e_1 : \texttt{num}$ and $\Gamma \vdash e_2 : \texttt{num}$ and $t = \texttt{bool}$.
- if $\Gamma \vdash \texttt{true} : t$, then $t = \texttt{bool}$.
- if $\Gamma \vdash \texttt{false} : t$, then $t = \texttt{bool}$.
- if $\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : t$,
  then $\Gamma \vdash e_b : \texttt{bool}$ and $\Gamma \vdash e_t : t$ and $\Gamma \vdash e_f : t$.
- if $\Gamma \vdash x : t$, then $\Gamma(x) = t$.
- if $\Gamma \vdash \texttt{fun}\, f(x : t_d) : t_r \triangleright e : t$,
  then $\Gamma, f : t_d \to t_r, x : t_d \vdash e : t_r$ and $t = t_d \to t_r$.
- if $\Gamma \vdash e_f\ e_a : t$, then $\Gamma \vdash e_f : t_d \to t_r$ and $\Gamma \vdash e_a : t_d$ and $t = t_r$.

Proof:

By induction on the typing rules.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash n : t$, then $t = \mathtt{num}$.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$ and $e = n$, then $t = \mathtt{num}$.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $e = n \Rightarrow t = \mathtt{num}$.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \forall n.\ e = n \Rightarrow t = \mathtt{num}$).

- Num:
  Show $\mathfrak{P}(\Gamma, n, \mathtt{num}) \equiv \forall n'.\ n = n' \Rightarrow \mathtt{num} = \mathtt{num}$.
  Trivial.
- Eq:
  Given $\mathfrak{P}(\Gamma, e_1, \mathtt{bool})$ and $\mathfrak{P}(\Gamma, e_2, \mathtt{bool})$.
  Show $\mathfrak{P}(\Gamma, e_1 + e_2, \mathtt{bool}) \equiv \forall n.\ e_1 + e_2 = n \Rightarrow \mathtt{bool} = \mathtt{num}$.
  Vacuous.
- Add, True, False, If, Var, Fun, App:
  Vacuous.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \forall n.\ e = n \Rightarrow t = \mathtt{num}$).

- Num:
  Show $\mathfrak{P}(\Gamma, n, \mathtt{num}) \equiv \forall n'.\ n = n' \Rightarrow \mathtt{num} = \mathtt{num}$.
  Trivial.
- Eq:
  Given $\mathfrak{P}(\Gamma, e_1, \mathtt{bool})$ and $\mathfrak{P}(\Gamma, e_2, \mathtt{bool})$.
  Show $\mathfrak{P}(\Gamma, e_1 + e_2, \mathtt{bool}) \equiv \forall n.\ e_1 + e_2 = n \Rightarrow \mathtt{bool} = \mathtt{num}$.
  Vacuous.
- Add, True, False, If, Var, Fun, App:
  Vacuous.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \forall n.\ e = n \Rightarrow t = \mathtt{num}$).

- Num:
  Show $\mathfrak{P}(\Gamma, n, \mathtt{num}) \equiv \forall n'.\ n = n' \Rightarrow \mathtt{num} = \mathtt{num}$.
  Trivial.
- Eq:
  Given $\mathfrak{P}(\Gamma, e_1, \mathtt{bool})$ and $\mathfrak{P}(\Gamma, e_2, \mathtt{bool})$.
  Show $\mathfrak{P}(\Gamma, e_1 + e_2, \mathtt{bool}) \equiv \forall n.\ e_1 + e_2 = n \Rightarrow \mathtt{bool} = \mathtt{num}$.
  Vacuous.
- Add, True, False, If, Var, Fun, App:
  Vacuous.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \forall n.\ e = n \Rightarrow t = \mathtt{num}$).

- Num:
  Show $\mathfrak{P}(\Gamma, n, \mathtt{num}) \equiv \forall n'.\ n = n' \Rightarrow \mathtt{num} = \mathtt{num}$.
  Trivial.
- Eq:
  Given $\mathfrak{P}(\Gamma, e_1, \mathtt{bool})$ and $\mathfrak{P}(\Gamma, e_2, \mathtt{bool})$.
  Show $\mathfrak{P}(\Gamma, e_1 + e_2, \mathtt{bool}) \equiv \forall n.\ e_1 + e_2 = n \Rightarrow \mathtt{bool} = \mathtt{num}$.
  Vacuous.
- Add,True,False,If,Var,Fun,App:
  Vacuous.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \forall n.\ e = n \Rightarrow t = \mathtt{num}$).

- Num:
  Show $\mathfrak{P}(\Gamma, n, \mathtt{num}) \equiv \forall n'.\ n = n' \Rightarrow \mathtt{num} = \mathtt{num}$.
  Trivial.
- Eq:
  Given $\mathfrak{P}(\Gamma, e_1, \mathtt{bool})$ and $\mathfrak{P}(\Gamma, e_2, \mathtt{bool})$.
  Show $\mathfrak{P}(\Gamma, e_1 + e_2, \mathtt{bool}) \equiv \forall n.\ e_1 + e_2 = n \Rightarrow \mathtt{bool} = \mathtt{num}$.
  Vacuous.
- Add, True, False, If, Var, Fun, App:
  Vacuous.

Properties of Typing

Lemma (Inversion; n):

- if $\Gamma \vdash e : t$, then $\forall n.\ e = n \Rightarrow t = \mathtt{num}$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \forall n.\ e = n \Rightarrow t = \mathtt{num}$).

- Num:
  Show $\mathfrak{P}(\Gamma, n, \mathtt{num}) \equiv \forall n'.\ n = n' \Rightarrow \mathtt{num} = \mathtt{num}$.
  Trivial.
- Eq:
  Given $\mathfrak{P}(\Gamma, e_1, \mathtt{bool})$ and $\mathfrak{P}(\Gamma, e_2, \mathtt{bool})$.
  Show $\mathfrak{P}(\Gamma, e_1 + e_2, \mathtt{bool}) \equiv \forall n.\ e_1 + e_2 = n \Rightarrow \mathtt{bool} = \mathtt{num}$.
  Vacuous.
- Add,True,False,If,Var,Fun,App:
  Vacuous.

Properties of Typing

Lemma (Weakening):
If $\Gamma \vdash e : t$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$, then $\Gamma' \vdash e : t$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$.

Intuition:
Extra "junk" in the context doesn't matter.

Properties of Typing

Lemma (Weakening):
If $\Gamma \vdash e : t$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$, then $\Gamma' \vdash e : t$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$.

Intuition:
Extra "junk" in the context doesn't matter.

Properties of Typing

Lemma (Weakening):
If $\Gamma \vdash e : t$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$, then $\Gamma' \vdash e : t$.

Proof:
By induction on the typing derivation $\Gamma \vdash e : t$.

Intuition:
Extra "junk" in the context doesn't matter.

Properties of Typing

Lemma (Substitution):
If $\Gamma \vdash e : t$ and $\Gamma(x) = t_x$ and $\Gamma \setminus x \vdash e_x : t_x$,
then $\Gamma \setminus x \vdash e[e_x/x] : t$.

Proof:
By induction on the typing derivation $\Gamma, x : t_x \vdash e : t$ (using Weakening).

Intuition:

$$\frac{\dfrac{\Gamma'(x) = t_x}{\Gamma' \vdash x : t_x} \quad \dfrac{\Gamma''(x) = t_x}{\Gamma'' \vdash x : t_x} \quad \begin{matrix} \vdots & \vdots \end{matrix}}{\Gamma \vdash e : t} \quad \Longrightarrow \quad \frac{\Gamma' \setminus x \vdash e_x : t_x \quad \Gamma'' \setminus x \vdash e_x : t_x \quad \begin{matrix} \vdots & \vdots \end{matrix}}{\Gamma \setminus x \vdash e[e_x/x] : t}$$

Properties of Typing

Lemma (Substitution):
If $\Gamma \vdash e : t$ and $\Gamma(x) = t_x$ and $\Gamma \setminus x \vdash e_x : t_x$,
then $\Gamma \setminus x \vdash e[e_x/x] : t$.

Proof:
By induction on the typing derivation $\Gamma, x : t_x \vdash e : t$ (using Weakening).

Intuition:

$$\frac{\dfrac{\Gamma'(x) = t_x}{\Gamma' \vdash x : t_x} \quad \dfrac{\Gamma''(x) = t_x}{\Gamma'' \vdash x : t_x} \quad \begin{matrix}\vdots & \vdots\end{matrix}}{\Gamma \vdash e : t} \quad \Longrightarrow \quad \frac{\Gamma' \setminus x \vdash e_x : t_x \quad \Gamma'' \setminus x \vdash e_x : t_x \quad \begin{matrix}\vdots & \vdots\end{matrix}}{\Gamma \setminus x \vdash e[e_x/x] : t}$$

Properties of Typing

Lemma (Substitution):
If $\Gamma \vdash e : t$ and $\Gamma(x) = t_x$ and $\Gamma \setminus x \vdash e_x : t_x$,
then $\Gamma \setminus x \vdash e[e_x/x] : t$.

Proof:
By induction on the typing derivation $\Gamma, x : t_x \vdash e : t$ (using Weakening).

Intuition:

$$\dfrac{\dfrac{\Gamma'(x) = t_x}{\Gamma' \vdash x : t_x} \quad \dfrac{\Gamma''(x) = t_x}{\Gamma'' \vdash x : t_x}}{\Gamma \vdash e : t} \quad \Longrightarrow \quad \dfrac{\Gamma' \setminus x \vdash e_x : t_x \quad \Gamma'' \setminus x \vdash e_x : t_x}{\Gamma \setminus x \vdash e[e_x/x] : t}$$

MinML

Dynamic Semantics of MinML

Dynamic Semantics

Describes how a program executes.

A number of different ways to specify:

- Denotational: Compile into a language (e.g., mathematics) with a well understood meaning.
- Axiomatic: Given some (logical) preconditions P, state the postconditions Q that hold after execution:
  - $\{P\}\ e\ \{Q\}$ – Hoare logic
- Operational: Define execution by rewriting the program step-by-step.
  - small-step
  - large-step
  - contextual

In this lecture, we give a small-step operational semantics.

Evaluation Judgements

An evaluation judgement is a tuple $e \longrightarrow e'$

- an expression e to evaluate, and
- an expression e' to which e evaluates (in one step).

Read: "The expression e steps to the expression e'."

A step consists of execution of a single "instruction".

Rules determine which "instruction" to execute next.

There are no steps from *values*.

Values

The values of MinML are defined by these rules:

$$\text{Values} \quad v \quad ::= \quad n \mid \texttt{true} \mid \texttt{false} \mid x \mid \texttt{fun}\, f(x : t_d) : t_r \triangleright e$$

Technically, the rules for the values of MinML define a judgement representing a *predicate* on expressions of MinML.

Using the metavariable v is more convenient
than (explicitly) defining the judgement e **isVal**
and adding $\cdots \wedge e$ **isVal** to rules and theorems.

Evaluation Rules

First, we define the primitive rules of MinML (the "instructions").

- Primitive operations on numbers.
- Conditional branch when the test is either `true` or `false`.
- Application of a recursive function to an argument value.

Second, we specify the next "instruction" to execute by search rules.
These rules specify the order of evaluation for MinML expressions.

- Left-to-right evaluation order.

Evaluation Rules – Primitive

The numeric operations have the expected primitive evaluation rules:

$$\frac{n_1 + n_2 = n}{n_1 + n_2 \longrightarrow n}\ \textsc{Add}$$

$$\frac{n_1 = n_2}{n_1 = n_2 \longrightarrow \texttt{true}}\ \textsc{EqT} \qquad \frac{n_1 \neq n_2}{n_1 = n_2 \longrightarrow \texttt{false}}\ \textsc{EqF}$$

Evaluation Rules – Primitive

Conditionals:

$$\frac{}{\texttt{if true then } e_t \texttt{ else } e_f \longrightarrow e_t}\ \textsc{IfT}$$

$$\frac{}{\texttt{if false then } e_t \texttt{ else } e_f \longrightarrow e_f}\ \textsc{IfF}$$

Evaluation Rules – Primitive

Applications:

$$\frac{}{(\texttt{fun}\, f(x : t_d) : t_r \triangleright e)\, v_a \longrightarrow e[(\texttt{fun}\, f(x : t_d) : t_r \triangleright e)/f][v_a/x]}\ \textsc{App}$$

- Substitute the entire recursive function expression for f in e.

Evaluation Rules – Search

The numeric operations are evaluated left-to-right:

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{AddL} \qquad \frac{e_2 \longrightarrow e_2'}{v_1 + e_2 \longrightarrow v_1 + e_2'}\ \textsc{AddR}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 = e_2 \longrightarrow e_1' = e_2}\ \textsc{EqL} \qquad \frac{e_2 \longrightarrow e_2'}{v_1 = e_2 \longrightarrow v_1 = e_2'}\ \textsc{EqR}$$

Evaluation Rules – Search

Conditionals search for the next "instruction" in the test expression:

$$\frac{e_b \longrightarrow e_b'}{\texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f \longrightarrow \texttt{if}\ e_b'\ \texttt{then}\ e_t\ \texttt{else}\ e_f}\ \textsc{If}$$

Evaluation Rules – Search

Applications are evaluated left-to-right:

$$\frac{e_f \longrightarrow e_f'}{e_f\ e_a \longrightarrow e_f'\ e_a}\ \textsc{AppF} \qquad \frac{e_a \longrightarrow e_a'}{v_f\ e_a \longrightarrow v_f\ e_a'}\ \textsc{AppA}$$

Multi-step Evaluation Judgements

The judgement $e \longrightarrow^* e''$ is inductively defined by the following rules:

$$\frac{}{e \longrightarrow^* e}\ \textsc{Refl} \qquad \frac{e \longrightarrow e' \qquad e' \longrightarrow^* e''}{e \longrightarrow^* e''}\ \textsc{Step}$$

Read: "The expression e multi-steps to the expression e''".

Intuitively, $e \longrightarrow^* e''$ iff $e \equiv e_0 \longrightarrow e_1 \longrightarrow \cdots \longrightarrow e_n \equiv e''$ for $n \geq 0$.

Evaluation Example

Consider the following expression/value v_{fact}:

$$\begin{array}{l}\mathtt{fun}\, f(n : \mathtt{num}) : \mathtt{num} \triangleright \\ \qquad \mathtt{if}\, n = 0\, \mathtt{then}\, 1\, \mathtt{else}\, n * (f\,(n-1))\end{array}$$

One step of evaluation:

$$\frac{}{v_{\mathsf{fact}}\, 3 \longrightarrow \mathtt{if}\, 3 = 0\, \mathtt{then}\, 1\, \mathtt{else}\, 3 * (v_{\mathsf{fact}}\,(3-1))}\ \textsc{App}$$

We have substituted v_{fact} for f and 3 for x in the body of the function.

Evaluation Example

$$
\begin{array}{rcl}
v_{\mathsf{fact}}\ 3 & \longrightarrow & \mathtt{if}\ 3 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * (v_{\mathsf{fact}}\ (3-1)) \\
& \longrightarrow & \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * (v_{\mathsf{fact}}\ (3-1)) \\
& \longrightarrow & 3 * (v_{\mathsf{fact}}\ (3-1)) \\
& \longrightarrow & 3 * (v_{\mathsf{fact}}\ 2) \\
& \longrightarrow & 3 * (\mathtt{if}\ 2 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 2 * (v_{\mathsf{fact}}\ (2-1))) \\
& \longrightarrow & \ldots \\
& \longrightarrow & 3 * (2 * (1 * 1)) \\
& \longrightarrow & 3 * (2 * 1) \\
& \longrightarrow & 3 * 2 \\
& \longrightarrow & 6
\end{array}
$$

- where

  $v_{\mathsf{fact}} = \mathtt{fun}\, f(n : \mathtt{num}) : \mathtt{num} \triangleright \mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ n * (f\ (n-1))$

Summary of Dynamic Semantics

The dynamic semantics of MinML is specified by inductive definitions of the evaluation judgements $e \longrightarrow e'$ and $e \longrightarrow^* e''$.

$$\mathfrak{D} \quad = \quad \frac{\vdots}{e \longrightarrow e'}$$

Properties of evaluation may be proved by induction (on evaluation derivations).

Induction on Evaluation Derivations

To prove that "If $e \longrightarrow e'$, then property $\mathfrak{P}(e, e')$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of (primitive rules):

- if $n_1 + n_2 = n$, then $\mathfrak{P}(n_1 + n_2, n)$
- if $n_1 = n_2$, then $\mathfrak{P}(n_1 = n_2, \texttt{true})$
- if $n_1 \neq n_2$, then $\mathfrak{P}(n_1 = n_2, \texttt{false})$
- $\mathfrak{P}(\texttt{if true then } e_t \texttt{ else } e_f, e_t)$
- $\mathfrak{P}(\texttt{if false then } e_t \texttt{ else } e_f, e_f)$
- $\mathfrak{P}((\texttt{fun } f(\mathcal{X} : t_d) : t_r \triangleright e)\, v_a, e[(\texttt{fun } f(\mathcal{X} : t_d) : t_r \triangleright e)/f][v_a/\mathcal{X}])$

Induction on Evaluation Derivations

To prove that "If $e \longrightarrow e'$, then property $\mathfrak{P}(e, e')$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of (search rules):

- if $\mathfrak{P}(e_1, e_1')$, then $\mathfrak{P}(e_1 + e_2, e_1' + e_2)$
- if $\mathfrak{P}(e_2, e_2')$, then $\mathfrak{P}(v_1 + e_2, v_1 + e_2')$
- if $\mathfrak{P}(e_1, e_1')$, then $\mathfrak{P}(e_1 = e_2, e_1' = e_2)$
- if $\mathfrak{P}(e_2, e_2')$, then $\mathfrak{P}(v_1 = e_2, v_1 = e_2')$
- if $\mathfrak{P}(e_b, e_b')$, then $\mathfrak{P}(\texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f, \texttt{if}\ e_b'\ \texttt{then}\ e_t\ \texttt{else}\ e_f)$
- if $\mathfrak{P}(e_f, e_f')$, then $\mathfrak{P}(e_f\ e_a, e_f'\ e_a)$
- if $\mathfrak{P}(e_a, e_a')$, then $\mathfrak{P}(v_f\ e_a, v_f\ e_a')$

Induction on Multi-step Evaluation Derivations

To prove that "If $e \longrightarrow^* e''$, then property $\mathfrak{Q}(e, e'')$ holds",
it is enough to show that the property holds for the conclusion of each rule given that it holds for each of the premises of the rule.

That is it is enough to show each of:

- $\mathfrak{Q}(e, e)$
- if $e \longrightarrow e'$ and $\mathfrak{Q}(e', e'')$, then $\mathfrak{Q}(e, e'')$
  - Often this involves proving (by induction) some property $\mathfrak{P}$ of the single-step evaluation derivation.

Properties of Evaluation

Lemma (Values Irreducible):
There is no e' such that $v \longrightarrow e'$.

Proof:
By induction on the evaluation rules.

Rephrase the lemma as "If $e \longrightarrow e'$, then $\forall v.\ e \neq v$".
By induction on the evaluation derivation $e \longrightarrow e'$.

Lingo: by inspection of the evaluation rules.

Properties of Evaluation

Lemma (Determinacy):
For every e there exists at most one e' such that $e \longrightarrow e'$.

Proof:
By induction on the structure of e (using Values Irreducible)

Lemma (Determinacy of Values):
For every e there exists at most one v'' such that $e \longrightarrow^* v''$.

Proof:
By induction on the derivation of $e \longrightarrow^* v''$ (using Determinacy).

Stuck States

Not every irreducible expression is a value!

- `if 7 then 1 else 2` does not reduce
- `true + false` does not reduce
- `true 1` does not reduce

If an expression is not a value and it doesn't reduce,
then its meaning is ill-defined.

- Anything can happen next

An expression e that is not a value and for which there exists no e' such that $e \longrightarrow e'$ is *stuck*.

Summary of Dynamic Semantics

The dynamic semantics of MinML is specified by inductive definitions of the evaluation judgements $e \longrightarrow e'$ and $e \longrightarrow^* e''$.

$$\mathfrak{D} \quad = \quad \frac{\vdots}{e \longrightarrow e'}$$

Evaluation is deterministic.

Evaluation can get stuck . . . if expressions are not well-typed.

Type Safety

A type system predicts something about the execution of a program.

$$\emptyset \vdash e : \mathtt{num} \longrightarrow \mathtt{num}$$

- the expression e will evaluate to a function value
  that takes an integer argument and returns an integer result,
  or does not terminate
- the expression e will not get stuck during evaluation

Type Safety

Type safety formalizes a coherence between the static and dynamic semantics of a programming language

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

The validity of the predictions guarantees that certain errors never occur. The kinds of error vary depending on the details of the type system.

- MinML predicts the form of valus (boolean? function? integer?)
- MinML guarentees that integers aren't applied to arguments

Formalization of Type Safety

Type safety formalizes a coherence between the static and dynamic semantics of a programming language

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

Theorem (Safety):
If $\emptyset \vdash e : t$ and $e \longrightarrow^* e'$, then e' is not stuck
(i.e., either e' is a value or there exists e'' such that $e' \longrightarrow e''$.

Formalization of Type Safety

Type safety formalizes a coherence between the static and dynamic semantics of a programming language

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

Theorem (Safety):
If $\emptyset \vdash e : t$ and $e \longrightarrow^* e'$, then e' is not stuck
(i.e., either e' is a value or there exists e'' such that $e' \longrightarrow e''$.

Proving Type Safety

Theorem (Safety):
If $\emptyset \vdash e : t$ and $e \longrightarrow^* e'$, then e' is not stuck
(i.e., either e' is a value or there exists e'' such that $e' \longrightarrow e''$.

A number of different ways to prove; we concentrate on a *syntactic* proof.

Prove type safety by showing two related properties:

- Preservation: A well-typed program remains well-typed during evaluation. (If an expression is well-typed and takes a step, then the stepped-to expression is well-typed.)
- Progress: A well-typed program is not stuck. (If an expression is well-typed, then it is either a value or it can take a step.)

Proving Type Safety

Theorem (Safety):
If $\emptyset \vdash e : t$ and $e \longrightarrow^* e'$, then e' is not stuck
(i.e., either e' is a value or there exists e'' such that $e' \longrightarrow e''$.

A number of different ways to prove; we concentrate on a *syntactic* proof.

Prove type safety by showing two related properties:

- Preservation: A well-typed program remains well-typed during evaluation. (If an expression is well-typed and takes a step, then the stepped-to expression is well-typed.)
- Progress: A well-typed program is not stuck. (If an expression is well-typed, then it is either a value or it can take a step.)

Proving Type Safety

Theorem (Safety):
If $\emptyset \vdash e : t$ and $e \longrightarrow^* e'$, then e' is not stuck
(i.e., either e' is a value or there exists e'' such that $e' \longrightarrow e''$.

Theorem (Preservation):
If $\emptyset \vdash e : t$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : t$.

Theorem (Progress):
If $\emptyset \vdash e : t$, then either e is a value or there exists e' such that $e \longrightarrow e'$.

Proving Preservation

Theorem (Preservation):
If $\emptyset \vdash e : t$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : t$.

Proof:
By induction on the evaluation derivation $e \longrightarrow e'$
(with $\mathfrak{P}(e, e') \equiv \forall t.\ \emptyset \vdash e : t \Rightarrow \emptyset \vdash e' : t$).

Proving Preservation

- ADD:

  $$\frac{n_1 + n_2 = n}{n_1 + n_2 \longrightarrow n}\ \textsc{Add}$$

  Have $n_1 + n_2 \longrightarrow n$

  and $\emptyset \vdash n_1 + n_2 : t$.

  Show $\emptyset \vdash n : t$.

  $\emptyset \vdash n_1 : \mathtt{num}$, $\emptyset \vdash n_2 : \mathtt{num}$, and $t = \mathtt{num}$

  (by Inversion lemma with $\emptyset \vdash n_1 + n_2 : t$)

  $\emptyset \vdash n : \mathtt{num}$

  (by NUM typing rule)

Proving Preservation

- ADD:

  $$\frac{n_1 + n_2 = n}{n_1 + n_2 \longrightarrow n}\ \textsc{Add}$$

  Have $n_1 + n_2 \longrightarrow n$

  and $\emptyset \vdash n_1 + n_2 : t$.

  Show $\emptyset \vdash n : t$.

  $\emptyset \vdash n_1 : \mathtt{num}$, $\emptyset \vdash n_2 : \mathtt{num}$, and $t = \mathtt{num}$

  (by Inversion lemma with $\emptyset \vdash n_1 + n_2 : t$)

  $\emptyset \vdash n : \mathtt{num}$

  (by NUM typing rule)

Proving Preservation

- ADD:

  $$\frac{n_1 + n_2 = n}{n_1 + n_2 \longrightarrow n}\ \textsc{Add}$$

  Have $n_1 + n_2 \longrightarrow n$

  and $\emptyset \vdash n_1 + n_2 : t$.

  Show $\emptyset \vdash n : t$.

  $\emptyset \vdash n_1 : \mathtt{num}$, $\emptyset \vdash n_2 : \mathtt{num}$, and $t = \mathtt{num}$

  (by Inversion lemma with $\emptyset \vdash n_1 + n_2 : t$)

  $\emptyset \vdash n : \mathtt{num}$

  (by NUM typing rule)

Proving Preservation

- App:

  $$\frac{}{v_f\ v_a \longrightarrow e[v_f/f][v_a/x]}\ \textsc{App}$$

  Have $v_f\ v_a \longrightarrow e[v_f/f][v_a/x]$
  and $\emptyset \vdash v_f\ v_a : t$,
  where $v_f = \mathtt{fun}\, f(x : t_d) : t_r \triangleright e$.
  Show $\emptyset \vdash e[v_f/f][v_a/x] : t$.

  $\emptyset \vdash v_f : t'_d \to t'_r$, $\emptyset \vdash v_a : t'_d$, and $t = t'_r$
  (by Inversion lemma with $\emptyset \vdash v_f\ v_a : t$)
  $\emptyset, f : t_d \to t_r, x : t_d \vdash e : t_r$ and $t'_d \to t'_r = t_d \to t_r$
  (by Inversion lemma with $\emptyset \vdash v_f : t'_d \to t'_r$)
  $\emptyset \vdash e[v_f/f][v_a/x] : t_r$ (by Substitution lemma (twice))

- Other primitive rules are similar.

Proving Preservation

- App:

  $$\frac{}{v_f\ v_a \longrightarrow e[v_f/f][v_a/x]}\ \textsc{App}$$

  Have $v_f\ v_a \longrightarrow e[v_f/f][v_a/x]$
  and $\emptyset \vdash v_f\ v_a : t$,
  where $v_f = \mathtt{fun}\, f(x : t_d) : t_r \triangleright e$.
  Show $\emptyset \vdash e[v_f/f][v_a/x] : t$.

  $\emptyset \vdash v_f : t'_d \to t'_r$, $\emptyset \vdash v_a : t'_d$, and $t = t'_r$
  (by Inversion lemma with $\emptyset \vdash v_f\ v_a : t$)

  $\emptyset, f : t_d \to t_r, x : t_d \vdash e : t_r$ and $t'_d \to t'_r = t_d \to t_r$
  (by Inversion lemma with $\emptyset \vdash v_f : t'_d \to t'_r$)

  $\emptyset \vdash e[v_f/f][v_a/x] : t_r$ (by Substitution lemma (twice))

- Other primitive rules are similar.

Proving Preservation

- App:

  $$\frac{}{v_f\ v_a \longrightarrow e[v_f/f][v_a/x]}\ \textsc{App}$$

  Have $v_f\ v_a \longrightarrow e[v_f/f][v_a/x]$
  and $\emptyset \vdash v_f\ v_a : t$,
  where $v_f = \mathtt{fun}\, f(x : t_d) : t_r \triangleright e$.
  Show $\emptyset \vdash e[v_f/f][v_a/x] : t$.

  $\emptyset \vdash v_f : t'_d \to t'_r$, $\emptyset \vdash v_a : t'_d$, and $t = t'_r$
  (by Inversion lemma with $\emptyset \vdash v_f\ v_a : t$)
  $\emptyset, f : t_d \to t_r, x : t_d \vdash e : t_r$ and $t'_d \to t'_r = t_d \to t_r$
  (by Inversion lemma with $\emptyset \vdash v_f : t'_d \to t'_r$)
  $\emptyset \vdash e[v_f/f][v_a/x] : t_r$ (by Substitution lemma (twice))

- Other primitive rules are similar.

Proving Preservation

- App:

  Have $\dfrac{}{v_f\ v_a \longrightarrow e[v_f/f][v_a/x]}\ \textsc{App}$

  and $\emptyset \vdash v_f\ v_a : t$,

  where $v_f = \mathtt{fun}\, f(x : t_d) : t_r \triangleright e$.

  Show $\emptyset \vdash e[v_f/f][v_a/x] : t$.

  $\emptyset \vdash v_f : t'_d \to t'_r$, $\emptyset \vdash v_a : t'_d$, and $t = t'_r$
  (by Inversion lemma with $\emptyset \vdash v_f\ v_a : t$)

  $\emptyset, f : t_d \to t_r, x : t_d \vdash e : t_r$ and $t'_d \to t'_r = t_d \to t_r$
  (by Inversion lemma with $\emptyset \vdash v_f : t'_d \to t'_r$)

  $\emptyset \vdash e[v_f/f][v_a/x] : t_r$ (by Substitution lemma (twice))

- Other primitive rules are similar.

Proving Preservation

- App:

  $$\frac{}{v_f\ v_a \longrightarrow e[v_f/f][v_a/x]}\ \textsc{App}$$

  Have $v_f\ v_a \longrightarrow e[v_f/f][v_a/x]$
  and $\emptyset \vdash v_f\ v_a : t$,
  where $v_f = \mathtt{fun}\, f(x : t_d) : t_r \triangleright e$.
  Show $\emptyset \vdash e[v_f/f][v_a/x] : t$.

  $\emptyset \vdash v_f : t'_d \to t'_r$, $\emptyset \vdash v_a : t'_d$, and $t = t'_r$
  (by Inversion lemma with $\emptyset \vdash v_f\ v_a : t$)
  $\emptyset, f : t_d \to t_r, x : t_d \vdash e : t_r$ and $t'_d \to t'_r = t_d \to t_r$
  (by Inversion lemma with $\emptyset \vdash v_f : t'_d \to t'_r$)
  $\emptyset \vdash e[v_f/f][v_a/x] : t_r$ (by Substitution lemma (twice))

- Other primitive rules are similar.

Proving Preservation

- ADDL:

  $$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{AddL}$$

  Have
  and $\emptyset \vdash e_1 + e_2 : t$
  and $\mathfrak{P}(e_1, e_1')$.
  Show $\emptyset \vdash e_1' + e_2 : t$.

  $\emptyset \vdash e_1 : \mathtt{num}$, $\emptyset \vdash e_2 : \mathtt{num}$, and $t = \mathtt{num}$
  (by Inversion lemma with $\emptyset \vdash e_1 + e_2 : t$)
  $\emptyset \vdash e_1' : \mathtt{num}$ (by induction hypothesis with $\emptyset \vdash e_1 : \mathtt{num}$)
  $\emptyset \vdash e_1' + e_2 : \mathtt{num}$ (by ADD typing rule)

- Other search rules are similar.

Proving Preservation

- AddL:

  $$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{AddL}$$

  Have
  and $\emptyset \vdash e_1 + e_2 : t$
  and $\mathfrak{P}(e_1, e_1')$.
  Show $\emptyset \vdash e_1' + e_2 : t$.

  $\emptyset \vdash e_1 : \mathtt{num}$, $\emptyset \vdash e_2 : \mathtt{num}$, and $t = \mathtt{num}$
  (by Inversion lemma with $\emptyset \vdash e_1 + e_2 : t$)
  $\emptyset \vdash e_1' : \mathtt{num}$ (by induction hypothesis with $\emptyset \vdash e_1 : \mathtt{num}$)
  $\emptyset \vdash e_1' + e_2 : \mathtt{num}$ (by Add typing rule)

- Other search rules are similar.

Proving Preservation

- ADDL:

  $$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{AddL}$$

  Have
  and $\emptyset \vdash e_1 + e_2 : t$
  and $\mathfrak{P}(e_1, e_1')$.
  Show $\emptyset \vdash e_1' + e_2 : t$.

  $\emptyset \vdash e_1 : \mathtt{num}$, $\emptyset \vdash e_2 : \mathtt{num}$, and $t = \mathtt{num}$
  (by Inversion lemma with $\emptyset \vdash e_1 + e_2 : t$)

  $\emptyset \vdash e_1' : \mathtt{num}$ (by induction hypothesis with $\emptyset \vdash e_1 : \mathtt{num}$)

  $\emptyset \vdash e_1' + e_2 : \mathtt{num}$ (by ADD typing rule)

- Other search rules are similar.

Proving Preservation

- AddL:

  $$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{AddL}$$

  Have
  and $\emptyset \vdash e_1 + e_2 : t$
  and $\mathfrak{P}(e_1, e_1')$.
  Show $\emptyset \vdash e_1' + e_2 : t$.

  $\emptyset \vdash e_1 : \mathtt{num}$, $\emptyset \vdash e_2 : \mathtt{num}$, and $t = \mathtt{num}$
  (by Inversion lemma with $\emptyset \vdash e_1 + e_2 : t$)
  $\emptyset \vdash e_1' : \mathtt{num}$ (by induction hypothesis with $\emptyset \vdash e_1 : \mathtt{num}$)
  $\emptyset \vdash e_1' + e_2 : \mathtt{num}$ (by Add typing rule)

- Other search rules are similar.

Proving Preservation

- AddL:

  $$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \textsc{AddL}$$

  Have
  and $\emptyset \vdash e_1 + e_2 : t$
  and $\mathfrak{P}(e_1, e_1')$.
  Show $\emptyset \vdash e_1' + e_2 : t$.

  $\emptyset \vdash e_1 : \mathtt{num}$, $\emptyset \vdash e_2 : \mathtt{num}$, and $t = \mathtt{num}$
  (by Inversion lemma with $\emptyset \vdash e_1 + e_2 : t$)
  $\emptyset \vdash e_1' : \mathtt{num}$ (by induction hypothesis with $\emptyset \vdash e_1 : \mathtt{num}$)
  $\emptyset \vdash e_1' + e_2 : \mathtt{num}$ (by Add typing rule)

- Other search rules are similar.

Proof of Preservation

How might the proof have failed?

Only if some evaluation step doesn't behave like its typing rule. e.g.:

$$\frac{n_1 = n_2}{n_1 = n_2 \longrightarrow 1}\ \textsc{EqT} \qquad \frac{n_1 \neq n_2}{n_1 = n_2 \longrightarrow 0}\ \textsc{EqF}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 = e_2 : \texttt{bool}}\ \textsc{Eq}$$

Preservation fails: In the case for $\textsc{EqT}$, would have to show $\emptyset \vdash 1 : \texttt{bool}$.

Proof of Preservation

Proof is by induction on evaluation derivation.

If an evaluation step is undefined, then Preservation still holds! e.g.:

$$\frac{n_1 = n_2}{n_1 = n_2 \longrightarrow \texttt{true}}\ \textsc{EqT}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 = e_2 : \texttt{bool}}\ \textsc{Eq}$$

Proving Progress

Theorem (Progress):
If $\emptyset \vdash e : t$, then either e is a value or there exists e' such that $e \longrightarrow e'$.

First, need a supporting lemma.

Proving Progress

The type of a closed value determines its form.

Lemma (Canonical Forms):
If $\emptyset \vdash v : t$, then

- if $t = \texttt{num}$, then $v = n$
- if $t = \texttt{bool}$, then $v = \texttt{true}$ or $v = \texttt{false}$
- if $t = t_d \rightarrow t_r$, then $v = \texttt{fun}\, f(x : t_d) : t_r \triangleright e$

Proof:
By induction on the derivation $\emptyset \vdash v : t$.

Proving Progress

Theorem (Progress):
If $\emptyset \vdash e : t$, then either e is a value or there exists e' such that $e \longrightarrow e'$.

Proof:
By induction on the typing derivation $\emptyset \vdash e : t$
(with $\mathfrak{P}(\Gamma, e, t) \equiv \Gamma = \emptyset \Rightarrow \exists v.\ e = v \vee \exists e'.\ e \longrightarrow e'$).

Proving Progress

- VAR:

  Have $$\frac{\emptyset(x) = t}{\emptyset \vdash x : t}\ \textsc{Var}$$.

  Vacuous.

Proving Progress

- VAR:

  Have $\dfrac{\emptyset(x) = t}{\emptyset \vdash x : t}\ \textsc{Var}$.

  Vacuous.

Proving Progress

- Num:

  Have $\dfrac{}{\emptyset \vdash n : \mathtt{num}}\ \textsc{Num}$.

  Show n is a value.

  Immediate (n is a value).

Proving Progress

- Num:

  Have $\dfrac{}{\emptyset \vdash n : \mathtt{num}}\ \textsc{Num}$.

  Show n is a value.

  Immediate (n is a value).

Proving Progress

- Num:

  Have $\dfrac{}{\emptyset \vdash n : \mathtt{num}}\ \textsc{Num}$.

  Show n is a value.

  Immediate (n is a value).

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.

Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)

$e_1 + e_2 \longrightarrow e_1' + e_2$ (by ADDL evaluation rule, if (2))

(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)

$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by ADDR evaluation rule, if (1) and (4))

$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))

$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))

$n_1 + n_1 \longrightarrow n$

(by ADD evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by AddL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by AddR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by Add evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by ADDL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by ADDR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by ADD evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by ADDL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by ADDR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by ADD evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by AddL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by AddR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by Add evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by AddL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by AddR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by Add evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by AddL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by AddR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by Add evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have $\emptyset \vdash e_1 + e_2 : \mathtt{num}$
and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by AddL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by AddR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by Add evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by AddL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by AddR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by Add evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

- ADD:

$$\frac{\emptyset \vdash e_1 : \mathtt{num} \qquad \emptyset \vdash e_2 : \mathtt{num}}{\emptyset \vdash e_1 + e_2 : \mathtt{num}}\ \textsc{Add}$$

Have $\emptyset \vdash e_1 + e_2 : \mathtt{num}$
and $\mathfrak{P}(\emptyset, e_1, \mathtt{num})$ and $\mathfrak{P}(\emptyset, e_2, \mathtt{num})$.
Show there exists e' such that $e_1 + e_2 \longrightarrow e'$.

(1) $e_1 = v_1$ or (2) $e_1 \longrightarrow e_1'$ (by induction hypothesis)
$e_1 + e_2 \longrightarrow e_1' + e_2$ (by ADDL evaluation rule, if (2))
(3) $e_2 = v_2$ or (4) $e_2 \longrightarrow e_2'$ (by induction hypothesis)
$v_1 + e_2 \longrightarrow v_1 + e_2'$ (by ADDR evaluation rule, if (1) and (4))
$v_1 = n_1$ (by Canonical Forms lemma with $\emptyset \vdash v_1 : \mathtt{num}$, if (1))
$v_2 = n_2$ (by Canonical Forms lemma with $\emptyset \vdash v_2 : \mathtt{num}$, if (3))
$n_1 + n_1 \longrightarrow n$
(by ADD evaluation rule with $n_1 + n_2 = n$, if (1) and (3))

Proving Progress

Cases for conditionals and applications are similar:

- use induction hypothesis to generate multiple cases
- use search rules when one sub-expression takes a step
- use Canonical Forms to show that primitive rules can be applied

Proof of Progress

How might the proof have failed?

Only if some evaluation step is undefined. e.g.:

$$\frac{n_1 = n_2}{n_1 = n_2 \longrightarrow \texttt{true}}\ \textsc{EqT}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 = e_2 : \texttt{bool}}\ \textsc{Eq}$$

Progress fails: In the case for $\textsc{Eq}$ with $e_1 = n_1$ and $e_2 = n_2$ and $n_1 \neq n_2$, we cannot take a step and $n_1 = n_2$ is not a value.

Proof of Progress

Proof is by induction on typing derivation.

If there is no typing rule for an expression form, then Progress still holds!

Summary

Type safety expresses the coherence of the static and operational semantics of a programming language.

A standard technique for proving type safety is via

- preservation
- progress

A type safety proof exhibits whether we have a sound language design and (if not) where to fix problems.