Structure of Programming Languages II: Types, Modules, Classes

Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

The Programming Language Core

- "Core" of values and operations establish fundamental capabilities of a language.
  - Numerical computation: numeric values.
  - Text editing: string values.
  - General purpose: core for many applications.
- Starting point for language design.
- Design programs and study their computational powers.
- Later extend core by conveniences.
  - Subroutines, modules, ...

Let's study the nature of a programming language core!

A Core Imperative Language

A while loop language

- Syntax domains.
- Syntax rules.

C $\in$ Command
E $\in$ Expression
L $\in$ Location
N $\in$ Numeral

C ::= L:=E | C_1;C_2 | **if** E **then** C_1 **else** C_2 **fi**
  | **while** E **do** C **od** | **skip**
E ::= N | @L | E_1+E_2 | ¬E | E_1=E_2
L ::= *loc*$_i$, if $i > 0$
N ::= n, if $n \in$ *Integer*

Example

loc$_1$:= 0; **while** @*loc*$_1$=0 **do** *loc*$_2$:=@*loc*$_1$+1 **od**

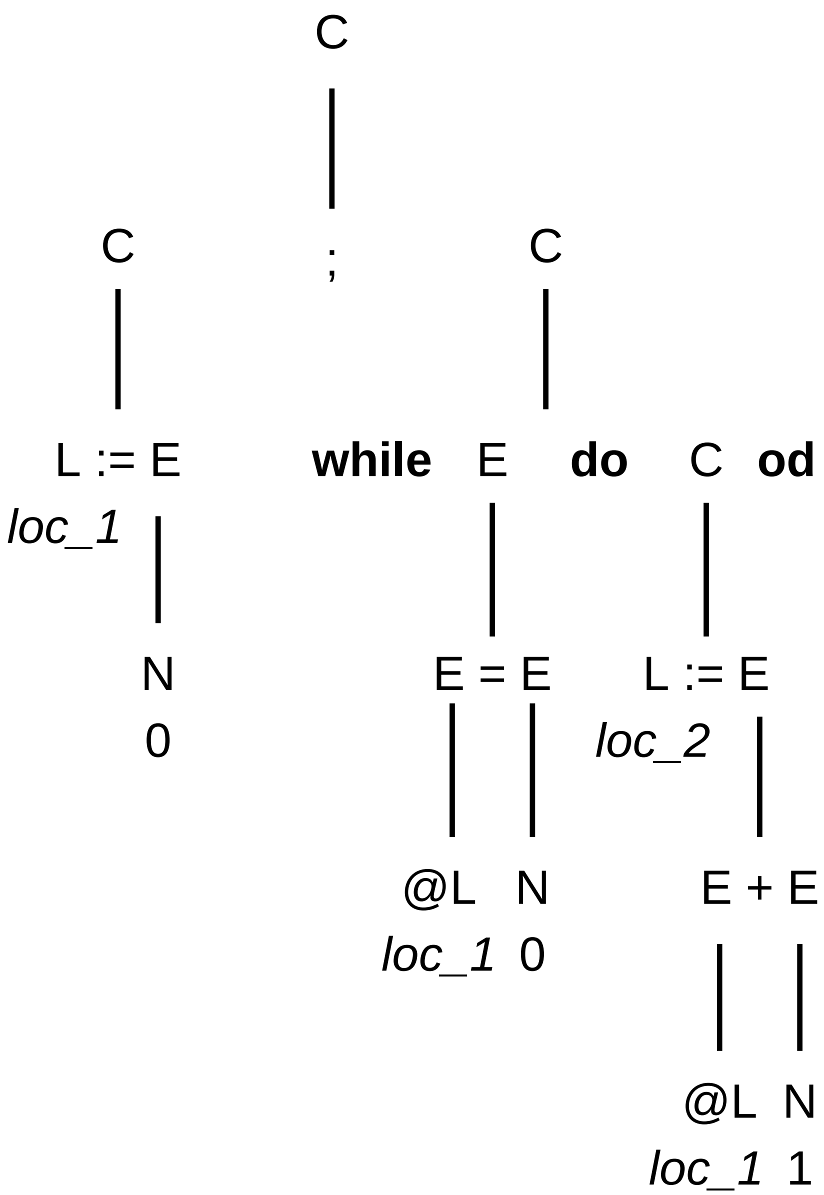
Abstract Syntax

- Non-terminal symbols.
  - C, E, L, N.
  - *Variables* over syntax trees.
- Terminal symbols.
  - @, +, :=, **skip**
  - *Labels* of syntax trees.
- *Inductive* definition of syntax trees.

Abstract syntax defines syntax trees!

Example

loc_1 := 0; **while** @loc_1=0 **do** loc_2:=@loc_1+1 **od**



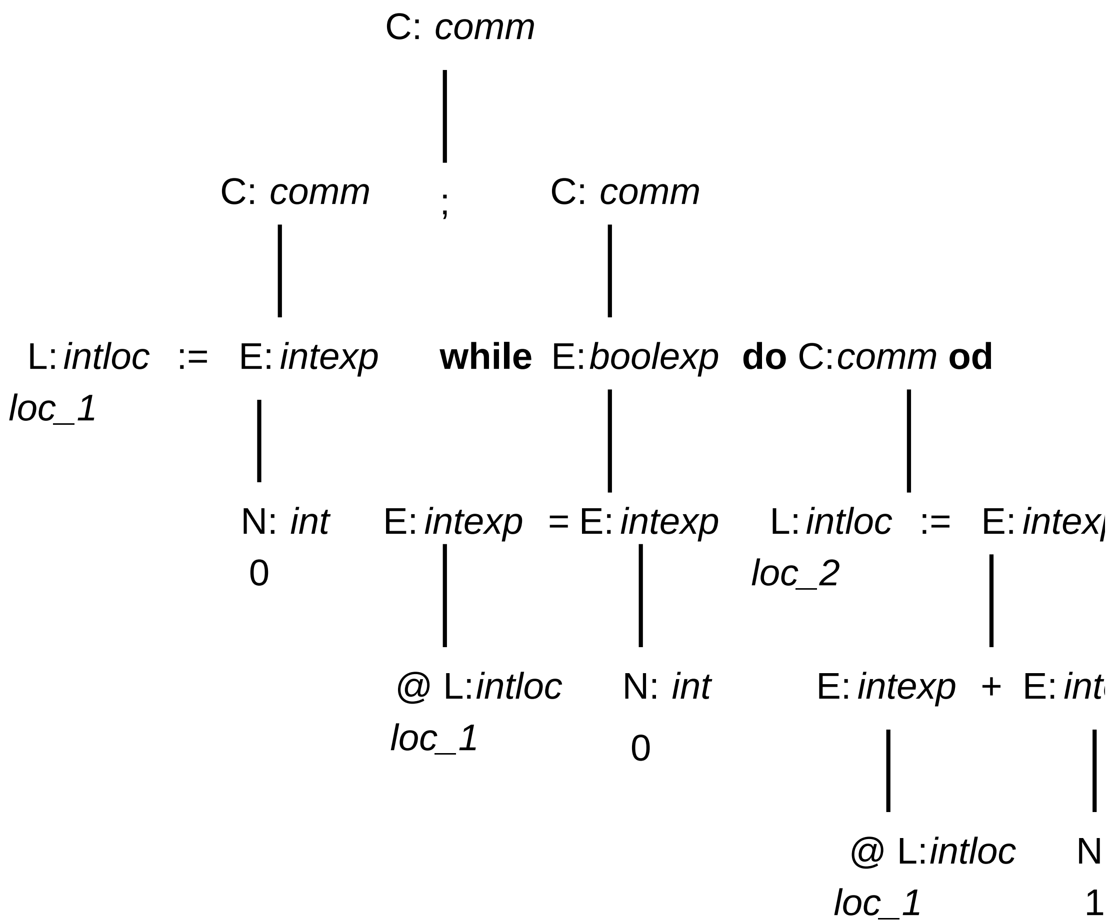
Semantics gives meaning to syntax trees!

Typing Rules

- Abstruct syntax does *not* define well-formed programs only.
  - Phrase "(0=1)+2" allowed.
  - Cannot add boolean to integer.
- Refine abstract syntax definition.
  - Integer and boolean expressions.
  - Define two distinct syntax domains?
  - Better: add typing annotations!
- *Attributed syntax trees*
  - Type attributes to all phrase forms.
  - Syntax tree is well typed if type attributes can be attached to all of its nonterminals.

Inference rules used for describing type structures.

Example



Each subtree is annotated with its type!

Typing Rules

Command

$$\frac{\text{L: } \textit{intloc} \quad \text{E: } \textit{intexp}}{\text{L:=E: } \textit{comm}} \qquad \frac{\text{C}_1\text{: } \textit{comm} \quad \text{C}_2\text{: } \textit{comm}}{\text{C}_1\text{;C}_2\text{: } \textit{comm}}$$

$$\frac{\text{E: } \textit{boolexp} \quad \text{C}_1\text{: } \textit{comm} \quad \text{C}_2\text{: } \textit{comm}}{\textbf{if } \text{E } \textbf{then } \text{C}_1 \textbf{ else } \text{C}_2 \textbf{ fi}\text{: } \textit{comm}}$$

$$\frac{\text{E: } \textit{boolexp} \qquad \text{C: } \textit{comm}}{\textbf{while } \text{E } \textbf{do } \text{C } \textbf{od}\text{: } \textit{comm}} \quad \textbf{skip}\text{: } \textit{comm}$$

Expression

$$\frac{\text{N: } \textit{int}}{\text{N: } \textit{intexp}} \qquad \frac{\text{L: } \textit{intloc}}{\text{@L: } \textit{intexp}}$$

$$\frac{\text{E}_1\text{: } \textit{intexp} \quad \text{E}_2\text{: } \textit{intexp}}{\text{E}_1\text{+E}_2\text{: } \textit{intexp}} \qquad \frac{\text{E: } \textit{boolexp}}{\neg\text{E: } \textit{boolexp}}$$

$$\frac{\text{E}_1\text{: } \tau\textit{exp} \quad \text{E}_2\text{: } \tau\textit{exp}}{\text{E}_1\text{=E}_2\text{: } \textit{boolexp}} \quad \text{if } \tau \in \{\textit{int}, \textit{bool}\}$$

Location

$\textit{loc}_i$: *intloc*, if $i > 0$

Numeral

n: *int*, if $n \in$ *Integer*

Typing Rules

- One typing rule for each construction of each syntax rule.
- Conditions under which constructions are well typed.
- Linear Notation (full type annotation):
  - $((\mathit{loc}_1)^{\mathit{intloc}}{:=}((0)^{\mathit{int}})^{\mathit{intexp}}$
    $(\mathbf{while}$
    $((@(\mathit{loc}_1)^{\mathit{intloc}})^{\mathit{intexp}}$
    $=((0)^{\mathit{int}})^{\mathit{intexp}})^{\mathit{boolexp}}$
    $(\mathbf{do}\ (\mathit{loc}_2)^{\mathit{intloc}}{:=}$
    $((@(\mathit{loc}_1)^{\mathit{intloc}})^{\mathit{intexp}}+$
    $((1)^{\mathit{int}})^{\mathit{intexp}})^{\mathit{intexp}})^{\mathit{comm}})^{\mathit{comm}})^{\mathit{comm}}.$
- Abbreviation (root type annotation):
  - $\mathit{loc}_1 := 0;$
    $\mathbf{while}\ @\mathit{loc}_1{=}0\ \mathbf{do}\ \mathit{loc}_2{:=}@\mathit{loc}_1{+}1\ \mathbf{od}\colon\ \mathit{comm}$

Typing Rules

- Logic assertion U:τ.
  - *Tree U is well typed with type τ.*
- *Static* typing for language.
  - Type attributes can be calculated without evaluating the program.
- *Strongly* typed language.
  - No run-time incompatibility errors.
- Unicity of typing.
  - Can a syntax tree be typed in multiple ways?
- Soundness of typing rules.
  - Are the typing rules sensible in their assignment of type attributes to phrases?

Questions will be addressed later.

Induction and Recursion

Syntax rule

$\mathsf{E} ::= \textbf{true} \mid \neg\mathsf{E} \mid \mathsf{E}_1\&\mathsf{E}_2$

- Inductive definition:
  - **true** is in Expression.
  - If E is in Expression, then so is $\neg$E.
  - If E_1 and E_2 are in Expression, then so is $E_1\&E_2$.
  - No other trees are in expression.
- Expression = set of trees!
- Generate all trees in stages.
  - $\mathit{stage}_0 = \{\}$.
  - $\mathit{stage}_{i+1} = \mathit{stage}_i \cup \{ \neg E \mid E \in \mathit{stage}_i\} \cup \{ E_1\&E_2 \mid E_1, E_2 \in \mathit{stage}_i\}$.
  - Expression $= \cup_{i\geq 0}\ \mathit{stage}_i$.
- Any tree in Expression is constructed in a finite number of stages.

Structural Induction

- Proof technique for syntax trees.
  - Goal: prove $P(t)$ for all trees t in a a language.
  - Inductive base: Prove that P holds for all trees in *stage*$_1$.
  - Inductive hypothesis: Assume that P holds for all trees in stages *stage*$_j$ with $j \leq i$.
  - Inductive step: Prove that P holds for all trees in *stage*$_i$.
- Prove $P(t)$ for all trees t in Expression.
  - P(**true**) holds.
  - P(¬E) holds assuming that P(E) holds (for arbitrary E).
  - $P(\mathrm{E}_1\&\mathrm{E}_2)$ holds assuming that $P(\mathrm{E}_1)$ holds and $P(\mathrm{E}_2)$ holds (for arbitrary E_1, E_2).

Syntax rules guide the proof!

Unicity of Typing

Can a syntax tree be typed in multiple ways?

- Unicity of typing property.
  - Every syntax tree has as most one assignment of typing attributes to its nodes.
  - If $P{:}\tau$ holds, then τ is unique.
- Unicity of Typing holds for Numeral.
  - By single typing rule, if $N{:}\tau$ holds, then $\tau = \textit{int}$ (for all N $\in$ Numeral).
- Unicity of Typing holds for Location.
  - By single typing rule, if $L{:}\tau$ holds, then $\tau = \textit{intloc}$ (for all L $\in$ Location).

Unicity of Typing

- Unicity of typing holds for Expression:
  - **Case** N. N:*int* holds. By single typing rule, N:*intexp* holds.
  - **Case** E_1+E_2. By inductive hypothesis, E_1:τ_1 and E_2:τ_2 hold for unique τ_1 and τ_2. By single typing rule, E_1:*intexp* and E_2:*intexp* must hold. If τ_1=τ_2 =*intexp*, then E_1+E_2:*intexp*. Otherwise, E_1+E_2 has no typing.
  - ...
- Unicity of typing holds for Command:
  - **Case** L := E. L:*intloc* holds. E:τ_1 holds for unique τ_1. By single typing rule, τ_1 must be *intexp* to have L:=E: *comm*. Otherwise, L:=E has no typing.
  - ...

Unicity of typing holds for all four syntax domains.

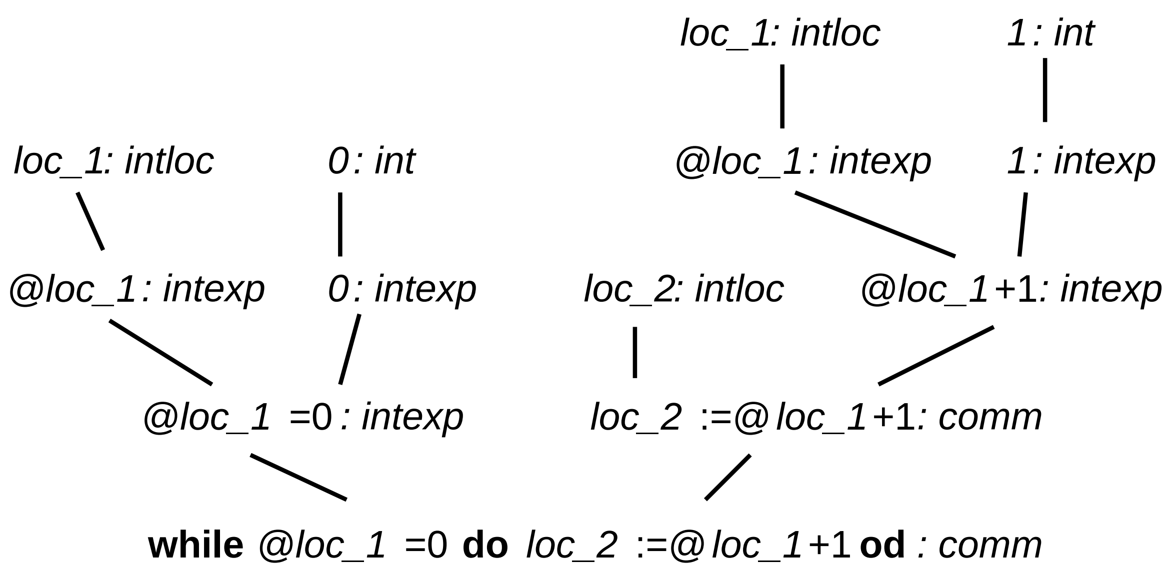
Typing Rules Define a Language

- *Typing rules* (not abstract syntax) define language.
  - Only well-formed programs are of value.
  - Programs are well-typed trees.
- Significance of unicity of typing:
  - Linear representation without type annotations represents (at most) one program.
  - Example: 0+1.
- Without unicity of typing:
  - *Coherence*: different tree derivations of a linear representation should have same meaning.
  - $\dfrac{\mathsf{E}{:}\textit{intexp}}{\mathsf{E}{:}\textit{realexp}} \qquad \dfrac{\mathsf{E}_1{:}\textit{realexp} \quad \mathsf{E}_2{:}\textit{realexp}}{\mathsf{E}_1{+}\mathsf{E}_2{:}\textit{realexp}}$
  - $(((0)^{\textit{int}}+(1)^{\textit{int}})^{\textit{intexp}})^{\textit{realexp}}$.
  - $((((0)^{\textit{int}})^{\textit{intexp}})^{\textit{realexp}} + (((1)^{\textit{int}})^{\textit{intexp}})^{\textit{realexp}})^{\textit{realexp}}$

Proof Trees

Programs may be directly derived from typing rules.

- Typing rules form a *logic*.
- Set of axioms and inference rules.
- (Inverted) trees are logic *proof trees*.



Semantics of the Core Language

Denotational semantics

- Recursively defined function.
  - Mapping of a well-typed derivation tree to its mathematical meaning.
- Semantic algebras.
  - Meaning sets (domains) and operations.
  - *Bool*, *Int*, *Location*, *Store*.
- For each typing rule, a recursive definition.
  - $\dfrac{\text{L: } \mathit{intloc} \quad \text{E: } \mathit{intexp}}{\text{L:=E: } \mathit{comm}}$
  - $[[\text{L:=E: } \mathit{comm}]] \ldots$
    $= \ldots [[\text{L: } \mathit{intloc}]] \ldots [[\text{E: } \mathit{intexp}]] \ldots$
- *Compositional* semantic definitions.
  - Meaning of tree constructed from meanings of its subtrees.

Function [[.]] is read as "the meaning of"

Semantic Algebras

$\mathit{Bool} = \{\mathit{true}, \mathit{false}\}$
$\mathit{not}: \mathit{Bool} \to \mathit{Bool}$
$\quad \mathit{not}(\mathit{false}) = \mathit{true}; \mathit{not}(\mathit{true}) = \mathit{false}$
$\mathit{equalbool}: \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}$
$\quad \mathit{equalbool}(m, n) = (m{=}n)$

$\mathit{Int} = \{\ldots, -1, 0, 1, \ldots\}$
$\mathit{plus}: \mathit{Int} \times \mathit{Int} \to \mathit{Int}$
$\quad \mathit{plus}(m, n) = m + n$
$\mathit{equalint}: \mathit{Int} \times \mathit{Int} \to \mathit{Bool}$
$\quad \mathit{equalint}(m, n) = (m{=}n)$

$\mathit{Location} = \{\mathit{loc}_i \mid i > 0\}$

Semantic Algebras

Store $= \{ \langle n_1, n_2, \ldots, n_m \rangle$
$\quad | \; n_i \in$ *Int*, $1 \leq i \leq m$, $m \geq 0 \}$
lookup: *Location* $\times$ *Store* $\rightarrow$ *Int*
$\quad$ *lookup*(*loc*$_j$, $\langle n_1, n_2, \ldots, n_j, \ldots, n_m \rangle) = n_j$
$\quad$ (if $j > m$, then *lookup*(*loc*$_j$, $\langle n_1, \ldots, n_m \rangle) = 0)$
update: *Location* $\times$ *Int* $\times$ *Store* $\rightarrow$ *Store*
$\quad$ *update*(*loc*$_j$, j, $\langle n_1, n_2, \ldots, n_j, \ldots, n_m \rangle) =$
$\qquad \langle n_1, n_2, \ldots, n, \ldots, n_m \rangle$
$\quad$ (if $j > m$, then *update*(*loc*$_j$, n, $\langle n_1, \ldots, n_m \rangle) =$
$\qquad \langle n_1, n_2, \ldots, n_m \rangle)$
if: *Bool* $\times$ *Store*$_\perp$ $\times$ *Store*$_\perp$ $\rightarrow$ *Store*$_\perp$
$\quad$ *if*(*true*, s_1, s_2) $= s_1$
$\quad$ *if*(*false*, s_1, s_2) $= s_2$
$\quad$ (*Store*$_\perp$ $=$ *Store* $\cup \{\perp\}$,
$\quad \perp =$ "bottom" $=$ non-termination)

Command Semantics

$[\![.\!: \textit{comm}]\!]: \textit{Store} \to \textit{Store}_\perp$

$[\![\text{L:=E}: \textit{comm}]\!](s) =$
$\quad \textit{update}([\![\text{L}: \textit{intloc}]\!], [\![\text{E}: \textit{intexp}]\!](s), s)$
$[\![\text{C}_1\text{;C}_2: \textit{comm}]\!](s) =$
$\quad [\![\text{C}_2: \textit{comm}]\!]([\![\text{C}_1: \textit{comm}]\!](s))$
$[\![\textbf{if } \text{E } \textbf{then } \text{C}_1 \textbf{ else } \text{C}_2 \textbf{ fi}: \textit{comm}]\!](s) =$
$\quad \textit{if}([\![\text{E}: \textit{boolexp}]\!](s),$
$\qquad [\![\text{C}_1: \textit{comm}]\!](s), [\![\text{C}_2: \textit{comm}]\!](s))$
$[\![\textbf{while } \text{E } \textbf{do } C \textbf{ od}: \textit{comm}]\!](s) = w(s)$
$\quad \text{where } w(s) =$
$\qquad \textit{if}([\![\text{E}: \textit{boolexp}]\!](s), w([\![\text{C}: \textit{comm}]\!](s)), s)$
$[\![\textbf{skip}: \textit{comm}]\!](s) = s$

The meaning of a command is a function from Store to Store.

Expression Semantics

$[\![.\text{: \_}\textit{exp}]\!]\text{: } \textit{Store} \rightarrow (\textit{Int} \cup \textit{Bool})$

$[\![\text{N: }\textit{intexp}]\!](s) = [\![\text{N: }\textit{int}]\!]$
$[\![\text{@L: }\textit{intexp}]\!](s) = \textit{lookup}([\![\text{L: }\textit{intloc}]\!], s)$
$[\![\neg\text{E: }\textit{boolexp}]\!](s) = \textit{not}([\![\text{E: }\textit{boolexp}]\!](s))$
$[\![\text{E}_1\text{+E}_2\text{: }\textit{intexp}]\!](s) =$
$\quad \textit{plus}([\![\text{E}_1\text{: }\textit{intexp}]\!](s), [\![\text{E}_2\text{: }\textit{intexp}]\!](s))$
$[\![\text{E}_1\text{=E}_2\text{: }\textit{boolexp}]\!](s) =$
$\quad \textit{equalbool}([\![\text{E}_1\text{: }\textit{boolexp}]\!](s), [\![\text{E}_2\text{: }\textit{boolexp}]\!](s))$
$[\![\text{E}_1\text{=E}_2\text{: }\textit{boolexp}]\!](s) =$
$\quad \textit{equalint}([\![\text{E}_1\text{: }\textit{intexp}]\!](s), [\![\text{E}_2\text{: }\textit{intexp}]\!](s))$

$[\![.\text{: }\textit{intloc}]\!]\text{: } \textit{Location}$

$[\![\textit{loc}_i\text{: }\textit{intloc}]\!] = \textit{loc}_i$

$[\![.\text{: }\textit{int}]\!]\text{: } \textit{Int}$

$[\![n\text{: }\textit{int}]\!] = n$

The meaning of an expression is a function from Store to Int or Bool.

Example

P = Q; R
Q = loc_2:=1; R = **if** @loc_2=0 **then skip else** S **fi**
S = loc_1:=@loc_2+4

[[P: *comm*]](s)
= [[R: *comm*]]([[Q: *comm*]](s))
= [[R: *comm*]]*update*(loc_2, 1, s)
= *if*([[@loc_2=0: *boolexp*]]*update*(loc_2, 1, s),
[[**skip**: *comm*]]*update*(loc_2, 1, s),
[[S: *comm*]]*update*(loc_2, 1, s))
= *if*(*false*, [[**skip**: *comm*]]*update*(loc_2, 1, s),
[[S: *comm*]]*update*(loc_2, 1, s))
= [[S: *comm*]]*update*(loc_2, 1, s)
= *update*(loc_1,
[[@loc_2+4: *intexp*]]*update*(loc_2, 1, s),
update(loc_2, 1, s))
= *update*(loc_1, 5, *update*(loc_2, 1, s))

Program semantics can be studied independently of specific storage vector!

Soundness of the Typing Rules.

Are the typing rules sensible in their assignment of type attributes to phrases?

- Typing rules must be *sound*.
  - Every well-typed program has a meaning.
- Type attributes:
  - $\tau ::= \mathit{int} \mid \mathit{bool}$
  - $\theta ::= \mathit{intloc} \mid \tau\,\mathit{exp} \mid \mathit{comm}$
- Mapping of attributes to meanings:
  - $[[\mathit{int}]] = \mathit{Int}$
  - $[[\mathit{bool}]] = \mathit{Bool}$
  - $[[\mathit{intloc}]] = \mathit{Location}$
  - $[[\tau\,\mathit{exp}]] = \mathit{Store} \rightarrow [[\tau]]$
  - $[[\mathit{comm}]] = \mathit{Store} \rightarrow \mathit{Store}_\perp$

How are [[P:θ]] and [[θ]] related?

Soundness Theorem

[[P:θ]] $\in$ [[θ]], for every well-typed phrase P:θ

- **Case** n: *int*
  - [[n: *int*]] $= n \in$ *Int* $=$ [[*int*]]
- **Case** @L: *intexp*
  - We know [[L: *intloc*]] $= l \in$ [[*intloc*]] $=$ *Location*. Then, for every Store s, [[@L: *intexp*]](s) $=$ *lookup*(l, s) $\in$ *Int* i.e. [[@L: *intexp*]] $\in$ *Store* $\rightarrow$ *Int* $=$ [[*intexp*]].
- **Case** C_1;C_2: *comm*
  - We know [[C_1: *comm*]] and [[C_2: *comm*]] are elements of Store $\rightarrow$ $Store_\perp$. For every Store s, we have [[C_1;C_2: *comm*]](s) $=$ [[C_2: *comm*]]([[C_1: *comm*]](s)) and [[C_2:*comm*]](s) $= s_1 \in$ *Store*$_\perp$. If s_1=$\perp$, [[C_2: *comm*]](s_1)=$\perp$ $\in$ *Store*$_\perp$. If $s_1 \in$ *Store*, then [[C_2: *comm*]](s_1) $\in$ *Store*$_\perp$. Hence, [[C_1;C_2: *comm*]] $\in$ *Store* $\rightarrow$ *Store*$_\perp$ $=$ [[*comm*]].

Prove one case for each typing rule.

Operational Properties

- Denotational semantics constructs *mathematical* functions.
  - Function extensionality for reasoning.
- Operational semantics reveals *computational* patterns.
  - Computation steps undertaken for evaluating the program.
- Denotational semantics has operational flower.
  - $[\![\mathit{loc}_3{:=}@\mathit{loc}_1{+}1\text{:}\ \mathit{comm}]\!] \langle 3,4,5\rangle$
    $\Rightarrow \mathit{update}(\mathit{loc}_3,$
    $\quad [\![@\mathit{loc}_1{+}1\text{:}\ \mathit{intexp}]\!] \langle 3,4,5\rangle,\ \langle 3,4,5\rangle)$
    $\Rightarrow \ldots$
    $\Rightarrow \mathit{update}(\mathit{loc}_3,\ 4,\ \langle 3,4,5\rangle)$
    $\Rightarrow \langle 3,4,4\rangle$

Can we use denotational definitions as operational rewrite rules?

Denotations as Rewrite Rules

Op. semantics reduces programs to values.

- A *program* is a phrase [[C: *comm*]]s_0.
- *Values* are from semantic domains.
  - Booleans, numerals, locations, storage vectors.
- Equational definitions get *rewrite rules*.
  - Denotation: $f(x_1, x_2, \ldots, x_n) = v$
  - Rule: $f(x_1, x_2, \ldots, x_n) \Rightarrow v$
- Computation is a sequence of rewrite steps $p_0 \Rightarrow^* p_n$.
  - $p_0 \Rightarrow p_1 \Rightarrow \ldots \Rightarrow p_n$.
  - Each computation step $p_i \Rightarrow p_{i+1}$ replaces a subphrase (the *redex*) in p_i according to some rewrite rule.
- If p_n is a value, computation *terminates*.

Which properties shall semantics fulfill?

Properties of Operational Semantics

- Soundness.
  - If p has an underlying "meaning" m and $p \Rightarrow p'$, then p' means m as well.
  - By definition of $\Rightarrow$.
- Subject reduction.
  - If p has an underlying "type" τ and $p \Rightarrow p'$, then p' has τ as well.
  - By soundness of typing.
- Strong typing.
  - If p is well-typed and $p \Rightarrow p'$, then p' contains no operator-operand incompatibilities.
  - By induction over computation rules.
- Computational adequacy.
  - A program p's underlying meaning is a proper meaning m, if there is some value v such that $p \Rightarrow v$ and v means m.

Computability of Phrases

- Predicate $\textit{comp}_\theta$ (*computable*):
  - $\textit{comp}_{\textit{intloc}}(p) := p \Rightarrow^* v$ and v means l where $l \in \textit{Location}$ is the meaning of p.
  - $\textit{comp}_{\tau\textit{exp}}(p) := p(s) \Rightarrow^* v$ and v means n where $s \in \textit{Store}$ and $p(s)$ means $n \in [[\tau]]$.
  - $\textit{comp}_{\textit{comm}}(p) := p(s) \Rightarrow^* v$ and v means s' where $s \in \textit{Store}$ and $p(s)$ means $s' \in \textit{Store}$.
- $\textit{comp}_\theta[[\mathsf{U}\colon\ \theta]]$ holds for all well-typed phrases $\mathsf{U}\colon \theta$.
  - Induction on typing rules.

Computational adequacy follows from soundness of typing, soundness of operational semantics and the computability of phrases.

Design of a Language Core

Contradictory design objectives:

- Oriented towards a specific problem area.
- General purpose.
- User friendly.
- Efficient implementation possible.
- Extensible by new language features.
- Secure agains programming errors.
- Simple syntax and semantics.
- Logical support for verification.
- ...

Design is an artistic activity!

Orthagonality

- A language should be based on few fundamental principles that may be combined without unneccessary restrictions.
- Orthagonal languages are easier to understand for programmers and implementors.
- Denotational semantics may help to achieve this.
  - Define sets of meanings and operations.
  - Give syntactic representations.
  - Organize into abstract syntax definition.

In the following we will study a set of basic design principles.