The Operational Semantics of Programs

a Component-Based Approach

L. Thomas van Binsbergen

15 March, 2017

Machines

- A *configuration* models the state of machine

$$\langle \mathit{assign}(x, 7), [x \mapsto 6], [1, 2, 3] \rangle$$

- **Abstract** *stored-program* machines
- The behaviour of a program is its effect on the machine

Configurations

- configuration = program component + auxiliary entities
- Auxiliary entities in this talk:
  - Store (σ), mapping identifiers ($\{x, y, \ldots\}$) to values ($\mathbb{Z} \cup \mathbb{B}$)
  - Output (α), a sequence of values

$$\langle \mathit{assign}(x, 7), [x \mapsto 6], [1, 2, 3] \rangle$$

Transitions

- Single transition written as $\gamma_1 \longrightarrow \gamma_2$
- A transition function 'modifies' configurations
- Computation = repeated transitions

(keep applying transition function until undefined)

$$\langle \mathit{seq}(\mathit{assign}(x, 1), \mathit{print}(x)), [\cdot], [\cdot] \rangle$$

- Configurations can be *terminal* or *stuck*
- Transitions have an *effect*, some transitions have *side effects*

Transitions

- Single transition written as $\gamma_1 \longrightarrow \gamma_2$
- A transition function 'modifies' configurations
- Computation = repeated transitions

(keep applying transition function until undefined)

$$\langle \mathit{seq}(\mathit{assign}(x, 1), \mathit{print}(x)), [\cdot], [\cdot] \rangle$$
$$\longrightarrow \langle \mathit{seq}(\mathit{done}, \mathit{print}(x)), [x \mapsto 1], [\cdot] \rangle$$

- Configurations can be *terminal* or *stuck*
- Transitions have an *effect*, some transitions have *side effects*

Transitions

- Single transition written as $\gamma_1 \longrightarrow \gamma_2$
- A transition function 'modifies' configurations
- Computation = repeated transitions

(keep applying transition function until undefined)

$$\begin{aligned} & \langle \mathit{seq}(\mathit{assign}(x, 1), \mathit{print}(x)), [\cdot], [\cdot] \rangle \\ \longrightarrow\ & \langle \mathit{seq}(\mathit{done}, \mathit{print}(x)), [x \mapsto 1], [\cdot] \rangle \\ \longrightarrow\ & \langle \mathit{print}(x), [x \mapsto 1], [\cdot] \rangle \end{aligned}$$

- Configurations can be *terminal* or *stuck*
- Transitions have an *effect*, some transitions have *side effects*

Transitions

- Single transition written as $\gamma_1 \longrightarrow \gamma_2$
- A transition function 'modifies' configurations
- Computation = repeated transitions

(keep applying transition function until undefined)

$$
\begin{aligned}
& \langle \mathit{seq}(\mathit{assign}(x, 1), \mathit{print}(x)), [\cdot], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{seq}(\mathit{done}, \mathit{print}(x)), [x \mapsto 1], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{print}(x), [x \mapsto 1], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{print}(1), [x \mapsto 1], [\cdot] \rangle
\end{aligned}
$$

- Configurations can be *terminal* or *stuck*
- Transitions have an *effect*, some transitions have *side effects*

Transitions

- Single transition written as $\gamma_1 \longrightarrow \gamma_2$
- A transition function 'modifies' configurations
- Computation = repeated transitions

(keep applying transition function until undefined)

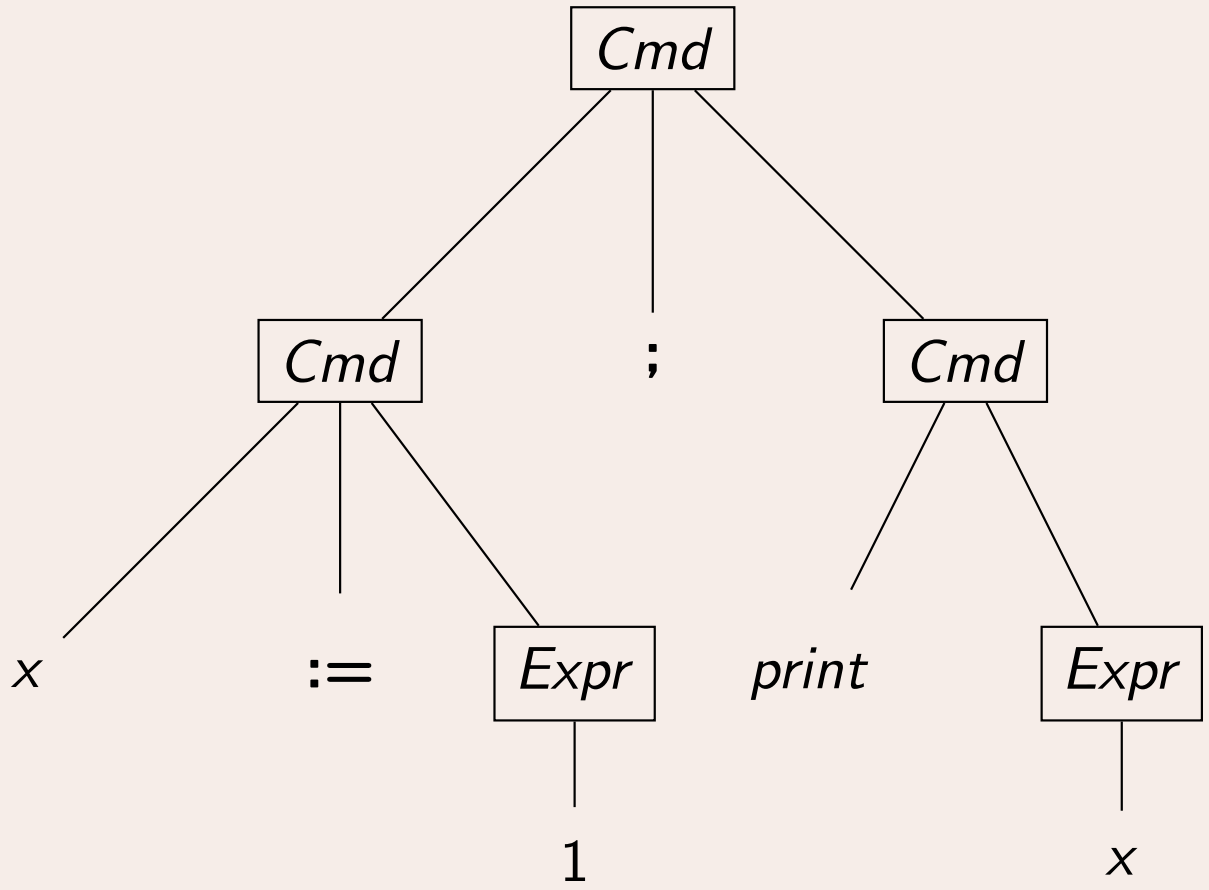
$$
\begin{aligned}
& \langle \mathit{seq}(\mathit{assign}(x, 1), \mathit{print}(x)), [\cdot], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{seq}(\mathit{done}, \mathit{print}(x)), [x \mapsto 1], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{print}(x), [x \mapsto 1], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{print}(1), [x \mapsto 1], [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{done}, [x \mapsto 1], [1] \rangle
\end{aligned}
$$

- Configurations can be *terminal* or *stuck*
- Transitions have an *effect*, some transitions have *side effects*

Concrete Program

$x := 1;$
print x

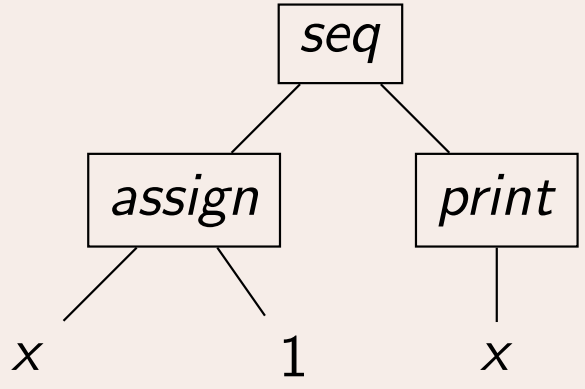
Parse Tree



Concrete Program

$x := 1;$
print x

Abstract Syntax Tree



Written as

$$seq(assign(x, 1), print(x))$$

Running example: **While**

Simple imperative language with the following *constructs*

Expressions - evaluate to a value without side-effects

- Literals: Booleans and integers ($\mathbb{B} \cup \mathbb{Z}$)
- Operators: *plus* and *leq*
- Looking up values (literals) assigned to identifiers ($\{x, y, \ldots\}$)

Commands - evaluate to *done* with potential side-effects

- Assignment
- One after the other
- While loops
- Later: printing

Abstract Syntax of **plus**

Informally

"An integer is an expression"

"A Boolean is an expression"

"Two expressions can be added together"

"If E_1 is an expression, and E_2 is an expression, then $plus(E_1, E_2)$ is an expression"

Formally

$$\mathbb{Z} \subseteq E$$

$$\mathbb{B} \subseteq E$$

$$\frac{E_1 \in E \qquad E_2 \in E}{plus(E_1, E_2) \in E}$$

Semantics of **plus** (1)

Informally

"If E_1 is an integer, and E_2 is an integer, then $plus(E_1, E_2)$ transitions to $E_1 + E_2$ *without any side-effects*"

Formally

$$\frac{E_1 \in \mathbb{Z} \qquad E_2 \in \mathbb{Z} \qquad I_1 = E_1 + E_2}{\langle plus(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle I_1, \sigma_1 \rangle}$$

What about for example $plus(plus(1, 2), plus(3, 4))$?

Semantics of **plus** (2)

Informally

"If E_1 is not a value, evaluate E_1 'within' *plus*"

"If E_1 has a transition to E_1', then $plus(E_1, E_2)$ transitions to $plus(E_1', E_2)$, *without any side-effects*"

Formally

$$\frac{\langle E_1, \sigma_1 \rangle \longrightarrow \langle E_1', \sigma_1 \rangle}{\langle plus(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle plus(E_1', E_2), \sigma_1 \rangle}$$

$$\langle plus(plus(1,2), plus(3,4)), [\cdot] \rangle \longrightarrow \langle plus(3, plus(3,4)), [\cdot] \rangle$$

Semantics of **plus** (3)

Informally

"If E_2 is not a value, evaluate E_2 'within' *plus*"
"If E_1 is a value, and E_2 is not a value, evaluate E_2 'within' *plus*"

"If E_1 is an integer, and E_2 has a transition to E_2', then *plus*(E_1, E_2) transitions to *plus*(E_1, E_2'), *without any side-effects*"

Formally

$$\frac{E_1 \in \mathbb{Z} \qquad \langle E_2, \sigma_1 \rangle \longrightarrow \langle E_2', \sigma_1 \rangle}{\langle \mathit{plus}(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle \mathit{plus}(E_1, E_2'), \sigma_1 \rangle}$$

$$\langle \mathit{plus}(3, \mathit{plus}(3, 4)), [\cdot] \rangle \longrightarrow \langle \mathit{plus}(3, 7), [\cdot] \rangle \longrightarrow \langle 10, [\cdot] \rangle$$

Example computation

Successful computation to terminal configuration

$$
\begin{aligned}
& \langle \mathit{plus}(\mathit{plus}(1,2), \mathit{plus}(3,4)), [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{plus}(3, \mathit{plus}(3,4)), [\cdot] \rangle \\
\longrightarrow\ & \langle \mathit{plus}(3,7), [\cdot] \rangle \\
\longrightarrow\ & \langle 10, [\cdot] \rangle
\end{aligned}
$$

Stuck configurations

$$\langle \mathit{plus}(\mathbf{true}, \mathit{plus}(1,2)), [\cdot] \rangle$$

$$\langle \mathit{plus}(1, \mathbf{false}), [x \mapsto 6] \rangle$$

$$\ldots$$

Semantics of *leq*

Abstract syntax rule

$$\frac{E_1 \in E \qquad E_2 \in E}{leq(E_1, E_2) \in E}$$

Transition rules

$$\frac{E_1 \in \mathbb{Z} \qquad E_2 \in \mathbb{Z} \qquad B_1 = E_1 \leqslant E_2}{\langle leq(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle B_1, \sigma_1 \rangle}$$

Semantics of *leq*

Abstract syntax rule

$$\frac{E_1 \in E \qquad E_2 \in E}{leq(E_1, E_2) \in E}$$

Transition rules

$$\frac{E_1 \in \mathbb{Z} \qquad \langle E_2, \sigma_1 \rangle \longrightarrow \langle E_2', \sigma_1 \rangle}{\langle leq(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle leq(E_1, E_2'), \sigma_1 \rangle}$$

$$\frac{E_1 \in \mathbb{Z} \qquad E_2 \in \mathbb{Z} \qquad B_1 = E_1 \leqslant E_2}{\langle leq(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle B_1, \sigma_1 \rangle}$$

Semantics of *leq*

Abstract syntax rule

$$\frac{E_1 \in E \qquad E_2 \in E}{leq(E_1, E_2) \in E}$$

Transition rules

$$\frac{\langle E_1, \sigma_1 \rangle \longrightarrow \langle E_1', \sigma_1 \rangle}{\langle leq(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle leq(E_1', E_2), \sigma_1 \rangle}$$

$$\frac{E_1 \in \mathbb{Z} \qquad \langle E_2, \sigma_1 \rangle \longrightarrow \langle E_2', \sigma_1 \rangle}{\langle leq(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle leq(E_1, E_2'), \sigma_1 \rangle}$$

$$\frac{E_1 \in \mathbb{Z} \qquad E_2 \in \mathbb{Z} \qquad B_1 = E_1 \leqslant E_2}{\langle leq(E_1, E_2), \sigma_1 \rangle \longrightarrow \langle B_1, \sigma_1 \rangle}$$

Semantics of identifiers

Abstract syntax rule

$$\{x, y, \ldots\} \subseteq E$$

Transition rules

$$\frac{Id_1 \in \{x, y, \ldots\} \qquad L_1 = \sigma_1(Id_1)}{\langle Id_1, \sigma_1 \rangle \longrightarrow \langle L_1, \sigma_1 \rangle}$$

Evaluating an expression

$$
\begin{aligned}
& \langle \mathit{leq}(\mathit{plus}(1,2), x), [x \mapsto 3, y \mapsto 6] \rangle \\
\longrightarrow\ & \langle \mathit{leq}(3, x), [x \mapsto 3, y \mapsto 6] \rangle \\
\longrightarrow\ & \langle \mathit{leq}(3, 3), [x \mapsto 3, y \mapsto 6] \rangle \\
\longrightarrow\ & \langle \mathbf{true}, [x \mapsto 3, y \mapsto 6] \rangle
\end{aligned}
$$

Semantics of **done**

Abstract syntax rule

$$done \in C$$

Transition rules

done is the result of executing a command, has no transition

Semantics of **assign**

Abstract syntax rule

$$\frac{Id_1 \in \{x, y, \ldots\} \qquad E_1 \in E}{assign(Id_1, E_1) \in C}$$

Transition rules

$$\frac{\langle E_1, \sigma_1 \rangle \longrightarrow \langle E_1', \sigma_1 \rangle}{\langle assign(Id_1, E_1), \sigma_1 \rangle \longrightarrow \langle assign(Id_1, E_1'), \sigma_1 \rangle}$$

$$\frac{L_1 \in \mathbb{Z} \cup \mathbb{B} \qquad \sigma_2 = \sigma_1[Id_1 \mapsto L_1]}{\langle assign(Id_1, L_1), \sigma_1 \rangle \longrightarrow \langle done, \sigma_2 \rangle}$$

Semantics of **seq**

Abstract syntax rule

$$\frac{C_1 \in C \qquad C_2 \in C}{seq(C_1, C_2) \in C}$$

Transition rules

$$\frac{\langle C_1, \sigma_1 \rangle \longrightarrow \langle C_1', \sigma_2 \rangle}{\langle seq(C_1, C_2), \sigma_1 \rangle \longrightarrow \langle seq(C_1', C_2), \sigma_2 \rangle}$$

$$\langle seq(done, C_2), \sigma_1 \rangle \longrightarrow \langle C_2, \sigma_1 \rangle$$

Semantics of **while**

Abstract syntax rule

$$\frac{E_1 \in E \qquad E_2 \in E \qquad C_1 \in C}{\mathit{while}(E_1, E_2, C_1) \in C}$$

Transition rules

$$\frac{\langle E_1, \sigma_1 \rangle \longrightarrow \langle E_1', \sigma_1 \rangle}{\langle \mathit{while}(E_1, E_2, C_1), \sigma_1 \rangle \longrightarrow \langle \mathit{while}(E_1', E_2, C_1), \sigma_1 \rangle}$$

$$\langle \mathit{while}(\mathbf{false}, E_2, C_1), \sigma_1 \rangle \longrightarrow \langle \mathit{done}, \sigma_1 \rangle$$

$$\langle \mathit{while}(\mathbf{true}, E_2, C_1), \sigma_1 \rangle \longrightarrow \langle \mathit{seq}(C_1, \mathit{while}(E_2, E_2, C_1)), \sigma_1 \rangle$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\langle while(CD, CD, BD), [x \mapsto 0] \rangle$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned} &\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\ \longrightarrow &\langle while(leq(0, 0), CD, BD), [x \mapsto 0]\rangle \end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0, 0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, 1), while(CD, CD, BD)), [x \mapsto 0]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, 1), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(done, while(CD, CD, BD)), [x \mapsto 1]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, 1), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(done, while(CD, CD, BD)), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(CD, CD, BD), [x \mapsto 1]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, 1), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(done, while(CD, CD, BD)), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(CD, CD, BD), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(leq(1,0), CD, BD), [x \mapsto 1]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, 1), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(done, while(CD, CD, BD)), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(CD, CD, BD), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(leq(1,0), CD, BD), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(\mathbf{false}, CD, BD), [x \mapsto 1]\rangle
\end{aligned}$$

Executing a command

$$CD = leq(x, 0)$$
$$BD = assign(x, plus(x, 1))$$

$$\begin{aligned}
&\langle while(CD, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(leq(0,0), CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle while(\mathbf{true}, CD, BD), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(BD, while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, plus(0,1)), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(assign(x, 1), while(CD, CD, BD)), [x \mapsto 0]\rangle \\
\longrightarrow &\langle seq(done, while(CD, CD, BD)), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(CD, CD, BD), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(leq(1,0), CD, BD), [x \mapsto 1]\rangle \\
\longrightarrow &\langle while(\mathbf{false}, CD, BD), [x \mapsto 1]\rangle \\
\longrightarrow &\langle done, [x \mapsto 1]\rangle
\end{aligned}$$

Semantics of **print**

Abstract syntax rule

$$\frac{E_1 \in E}{print(E_1) \in C}$$

Transition rules

$$\frac{\langle E_1, \sigma_1, \alpha_1 \rangle \longrightarrow \langle E_1', \sigma_1, \alpha_1 \rangle}{\langle print(E_1), \sigma_1, \alpha_1 \rangle \longrightarrow \langle print(E_1'), \sigma_1, \alpha_1 \rangle}$$

$$\frac{L_1 \in \mathbb{Z} \cup \mathbb{B} \qquad \alpha_2 = \alpha_1 +\!\!+ [L_1]}{\langle print(L_1), \sigma_1, \alpha_1 \rangle \longrightarrow \langle done, \sigma_1, \alpha_2 \rangle}$$

$$\langle print(plus(1,2)), [\cdot], [\cdot] \rangle \longrightarrow \langle print(3), [\cdot], [\cdot] \rangle \longrightarrow \langle done, [\cdot], [3] \rangle$$

The PLanCompS Approach

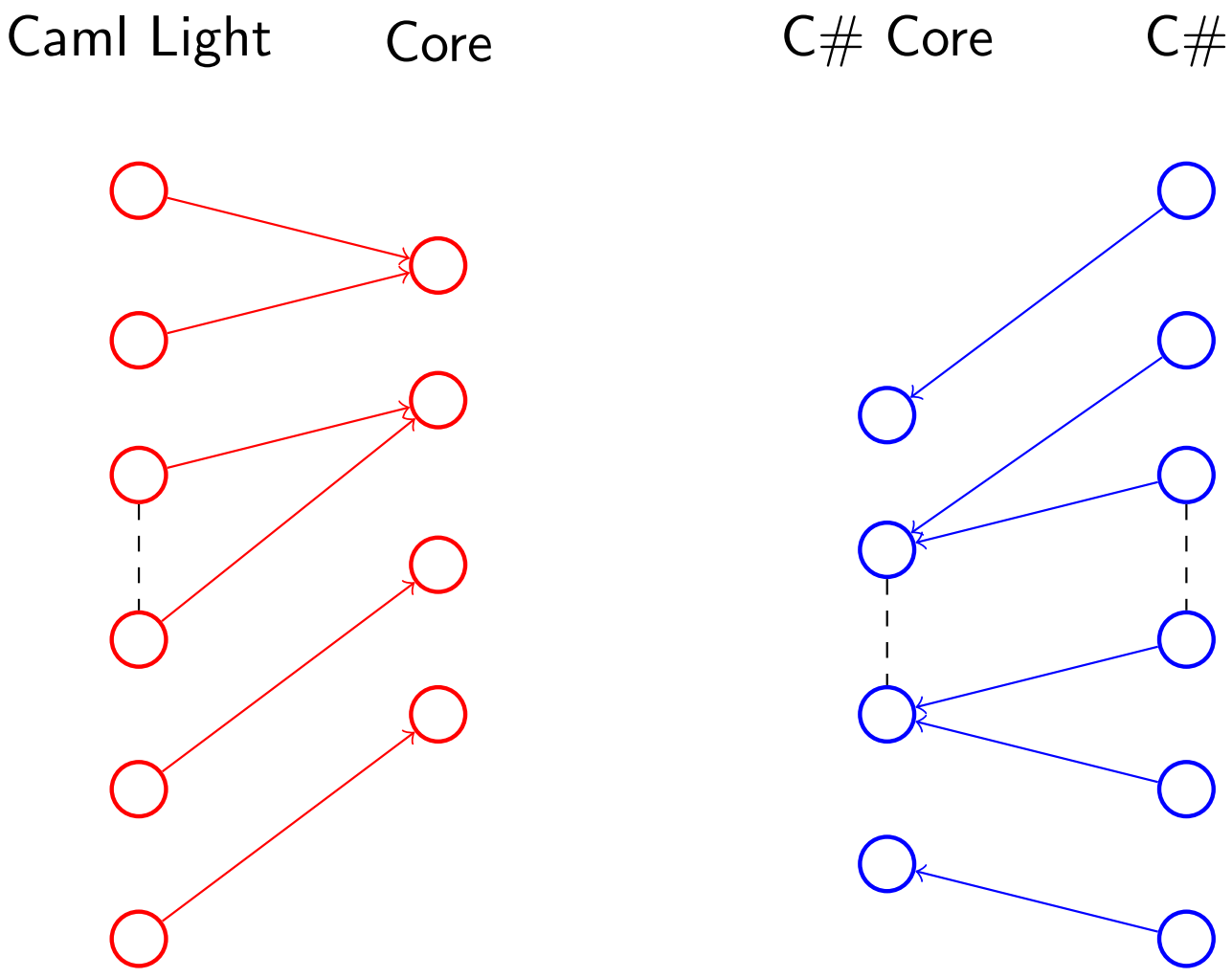
PLanCompS project (2011-2015)

- Swansea University
- Royal Holloway, University of London
- http://plancomps.org

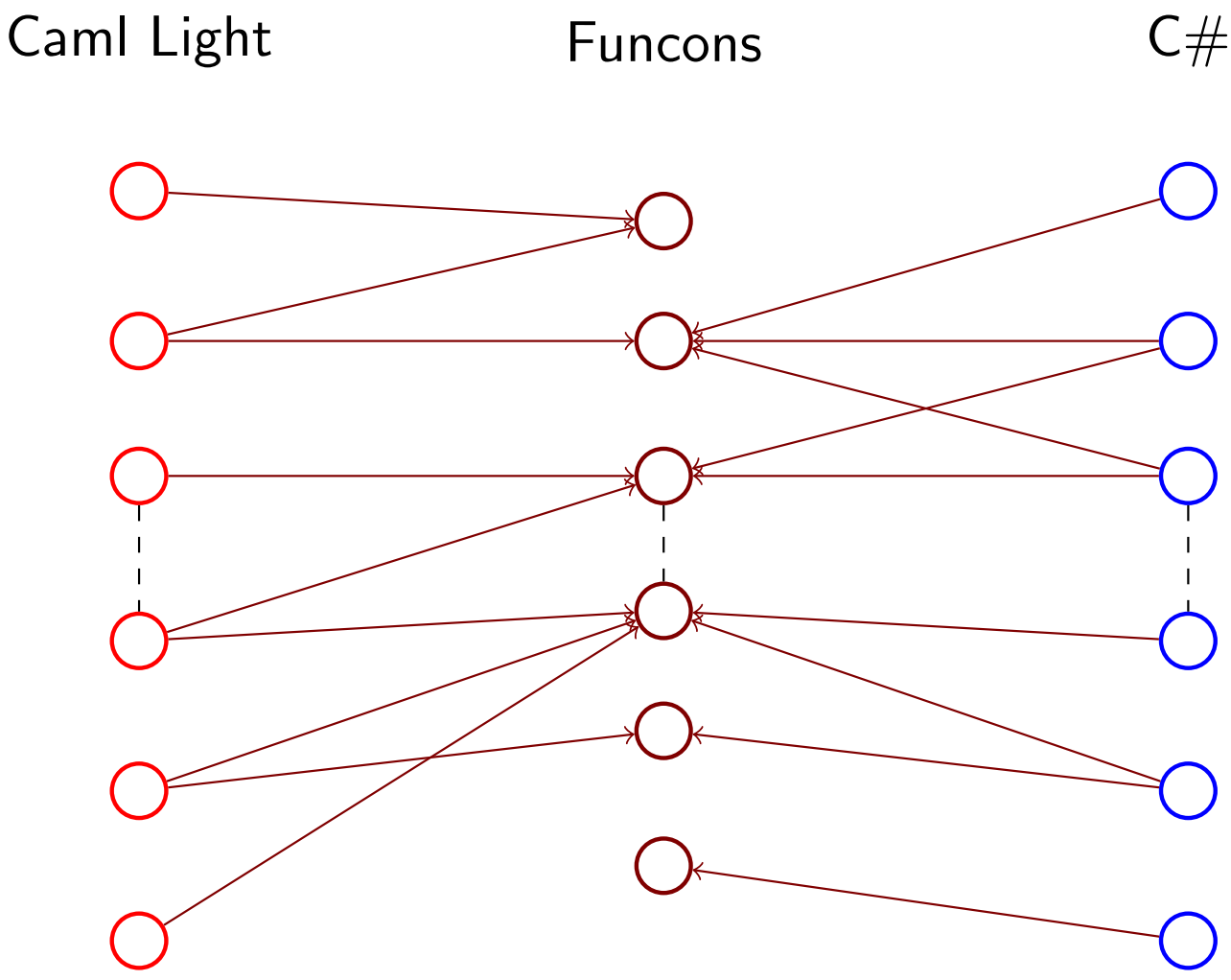
Approach

- Component based approach towards formal semantics
- Highly reusable, *fun*damental *con*structs: *funcons*
- Each funcon has a formal definition
- A language is defined formally via a translation to funcons

Reusable Components: Funcons



Reusable Components: Funcons



Executable Language Definitions

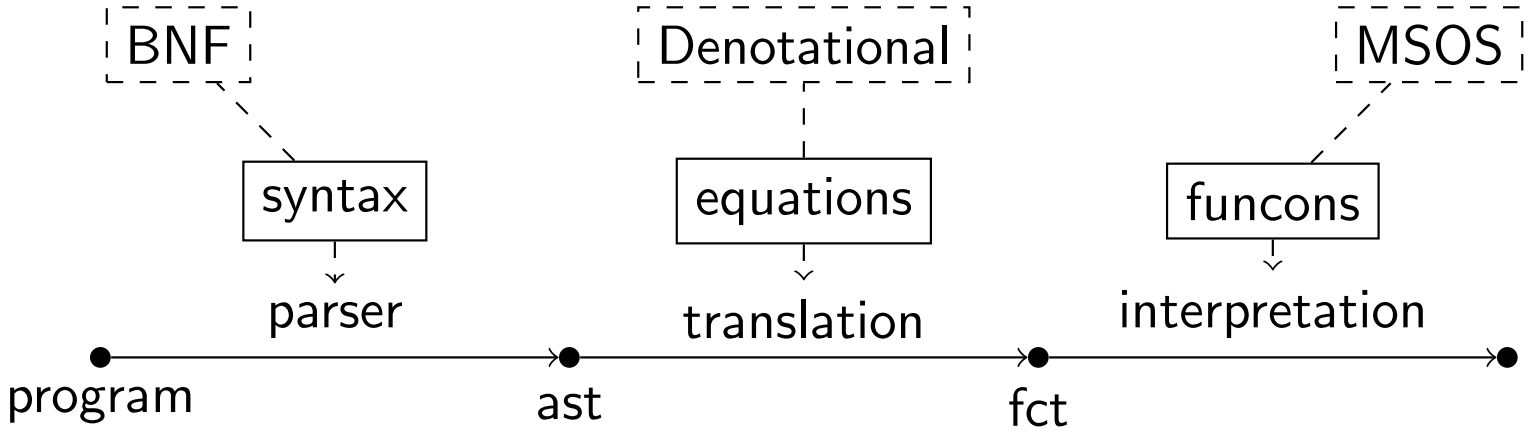


Figure : PLanCompS: generate interpreters from reusable specification.

Component-based Semantics for WHILE

$$
\begin{aligned}
[\![\textbf{true}]\!] &= \textbf{true} \\
[\![\textbf{false}]\!] &= \textbf{false} \\
[\![I_1]\!] &= I_1 && (I_1 \in \mathbb{Z}) \\
[\![\mathit{plus}(E_1, E_2)]\!] &= \textbf{sum}([\![E_1]\!], [\![E_2]\!]) \\
[\![\mathit{leq}(E_1, E_2)]\!] &= \textbf{is-less-or-equal}([\![E_1]\!], [\![E_2]\!]) \\
[\![\mathit{Id}_1]\!] &= \textbf{assigned}([\![\mathit{Id}_1]\!]) && (\mathit{Id}_1 \in \{x, y, \ldots\})
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathit{assign}(\mathit{Id}_1, E_1)]\!] &= \textbf{assign}(\mathit{Id}_1, [\![E_1]\!]) \\
[\![\mathit{seq}(C_1, C_2)]\!] &= \textbf{sequential}([\![C_1]\!], [\![C_2]\!]) \\
[\![\mathit{while}(E_1, \_, C_1)]\!] &= \textbf{while}([\![E_1]\!], [\![C_1]\!]) \\
[\![\mathit{print}(E_1)]\!] &= \textbf{print}([\![E_1]\!]) \\
[\![\mathit{done}]\!] &= \textbf{void}
\end{aligned}
$$

Record Configurations

Configurations

$$\langle P, \sigma, \alpha \rangle$$

"We mean the specific configuration that is a triple with first element P, second element σ, and third element α"

Record Configurations

$$\{pc = P, store = \sigma, out = \alpha\}$$

"We mean any configuration that has at least the program component (pc) P, the store σ and the output (out) α"

Record Configurations

Configurations

$$\langle P, \sigma, \alpha \rangle$$

"We mean the specific configuration that is a triple with first element P, second element σ, and third element α"

Record Configurations

$$\{P, store = \sigma, out = \alpha\}$$

"We mean any configuration that has at least the program component (pc) P, the store σ and the output (out) α"

Record Transitions

Not quite right

$$\{\textbf{sum}(1,2)\} \longrightarrow \{3\}$$

"*Any configuration with program component* **sum**$(1,2)$ *transitions into any configuration with program component* 3"

- Valid instantiation: $\langle \textbf{sum}(1,2), [\cdot], [\cdot] \rangle \longrightarrow \langle 3, [\cdot], [666] \rangle$

Remember the informal semantics of *plus*

"If E_1 is an integer, and E_2 is an integer, then $plus(E_1, E_2)$ transitions to $E_1 + E_2$ *without any side-effects*"

- We need to constrain the instantiations of the records!
- Based on the number of *premises* in a rule

Modular Inference Rules (Axioms)

Implicit

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1]}{\{\textbf{print}(V_1), \mathit{out} = \alpha_1\} \longrightarrow \{\textbf{void}, \mathit{out} = \alpha_2\}}$$

Modular Inference Rules (Axioms)

Implicit

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1]}{\{\textbf{print}(V_1), \mathit{out} = \alpha_1\} \longrightarrow \{\textbf{void}, \mathit{out} = \alpha_2\}}$$

Explicit

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1] \qquad X = Y}{\{\textbf{print}(V_1), \mathit{out} = \alpha_1\} \cup X \longrightarrow \{\textbf{void}, \mathit{out} = \alpha_2\} \cup Y}$$

Modular Inference Rules (Axioms)

Implicit

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1]}{\{\textbf{print}(V_1), out = \alpha_1\} \longrightarrow \{\textbf{void}, out = \alpha_2\}}$$

Explicit

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1] \qquad X = Y}{\{\textbf{print}(V_1), out = \alpha_1\} \cup X \longrightarrow \{\textbf{void}, out = \alpha_2\} \cup Y}$$

Valid Judgement

$$\langle \textbf{print}(3), [x \mapsto \textbf{true}], [\cdot] \rangle \longrightarrow \langle \textbf{void}, [x \mapsto \textbf{true}], [3] \rangle$$

Modular Inference Rules (1 premise)

Implicit

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{print}(F_1)\} \longrightarrow \{\textbf{print}(F_1')\}}$$

Modular Inference Rules (1 premise)

Implicit

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{print}(F_1)\} \longrightarrow \{\textbf{print}(F_1')\}}$$

Explicit

$$\frac{\{F_1\} \cup X \longrightarrow \{F_1'\} \cup Y}{\{\textbf{print}(F_1)\} \cup X \longrightarrow \{\textbf{print}(F_1')\} \cup Y}$$

Modular Inference Rules (1 premise)

Implicit

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{print}(F_1)\} \longrightarrow \{\textbf{print}(F_1')\}}$$

Explicit

$$\frac{\{F_1\} \cup X \longrightarrow \{F_1'\} \cup Y}{\{\textbf{print}(F_1)\} \cup X \longrightarrow \{\textbf{print}(F_1')\} \cup Y}$$

Valid Judgement

$$\langle \textbf{print}(x), [x \mapsto 1], [\cdot] \rangle \longrightarrow \langle \textbf{print}(1), [x \mapsto 1], [\cdot] \rangle$$

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{print}(F_1)\} \longrightarrow \{\textbf{print}(F_1')\}} \quad (1)$$

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1]}{\{\textbf{print}(V_1), out = \alpha_1\} \longrightarrow \{\textbf{void}, out = \alpha_2\}} \quad (2)$$

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{assign}(Id_1, F_1)\} \longrightarrow \{\textbf{assign}(Id_1, F_1')\}} \quad (3)$$

$$\frac{\texttt{is-value}(V_1) \qquad \sigma_2 = \sigma_1[Id_1 \mapsto V_1]}{\{\textbf{assign}(Id_1, V_1), store = \sigma_1\} \longrightarrow \{\textbf{void}, store = \sigma_2\}} \quad (4)$$

$$\langle \textbf{assign}(x, \textbf{print}(3)), [\cdot], [\cdot] \rangle$$
$$\rightarrow \langle \textbf{assign}(x, \textbf{void}), [\cdot], [3] \rangle$$

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{print}(F_1)\} \longrightarrow \{\textbf{print}(F_1')\}} \quad (1)$$

$$\frac{\texttt{is-value}(V_1) \qquad \alpha_2 = \alpha_1 +\!\!+ [V_1]}{\{\textbf{print}(V_1), out = \alpha_1\} \longrightarrow \{\textbf{void}, out = \alpha_2\}} \quad (2)$$

$$\frac{\{F_1\} \longrightarrow \{F_1'\}}{\{\textbf{assign}(Id_1, F_1)\} \longrightarrow \{\textbf{assign}(Id_1, F_1')\}} \quad (3)$$

$$\frac{\texttt{is-value}(V_1) \qquad \sigma_2 = \sigma_1[Id_1 \mapsto V_1]}{\{\textbf{assign}(Id_1, V_1), store = \sigma_1\} \longrightarrow \{\textbf{void}, store = \sigma_2\}} \quad (4)$$

$$\begin{aligned} &\langle \textbf{assign}(x, \textbf{print}(3)), [\cdot], [\cdot] \rangle \\ \rightarrow &\langle \textbf{assign}(x, \textbf{void}), [\cdot], [3] \rangle \\ \rightarrow &\langle \textbf{void}, [x \mapsto \textbf{void}], [3] \rangle \end{aligned}$$

Conclusions

We have seen:

- Transition functions for defining operational semantics
- Inference rules defining abstract syntax *and* transition functions
- That transition rules can be written in a modular fashion
- How a component-based semantics of a language can be given

"**divide and conquer** *is applicable to the formal semantics of programming languages*"

We have not seen that:

- Component-based semantics are executable
- Case studies suggest the approach is practical
- Scales to languages with:
  objects, exceptions, delimited continuations, etc.

The Operational Semantics of Programs

a Component-Based Approach

L. Thomas van Binsbergen

15 March, 2017

L. Thomas van Binsbergen

15 March, 1989