Modelos Formales de Computación

Part 2: Semantics of Programming Languages

Máster en Ingeniería y Tecnología de Sistemas Software

Outline

Outline

The objectives

- Know the different approaches for semantics in the literature
- Understand the meaning of the function $[\![\_]\!]$
- Understand the fixpoint computations for loops
- Understand the restricted evaluation of if-then-else
- Compare the different semantics using a programming language (SIMPLE)

Outline

Programming languages definitions

- Programming languages are usually presented quite informally, in large-audience books and manuals, some of them called, for some reason, “idiot’s guides”.
- Some programming languages have a standardization committee, e.g. the language C specified by an ISO standard, similarly Ada, Fortran, Pascal, or Ruby.
- Quite often a programming language is simple defined by one or two “reference” compilers or interpreters, e.g. Perl, Prolog, Visual Basic.
- While books, standards or reference implementations have clearly the merit of making programming languages quickly available to masses of programmers, they cannot serve as “official” definitions of languages.

Errors in Program Semantics

Consider this program (unseq.c):

```
int main() {
        int x = 0;
        return (x = 1) + (x = 2);
}
```

When compiled with clang, this program returns 3 but with gcc we unexpectedly get 4. We can see

C. Hathhorn, G. Rosu: Dealing With C's Original Sin. IEEE Software. 36(5): 24-28 (2019)

Formal Semantics

- From the point of view of a formally untrained software engineer, formal definitions of languages may look slightly too cryptic.
- The role of mathematical theories in general is two-fold:
  - they help understanding the defined concept better, and
  - are ameanable to formal, mechanically certifiable proofs.

  In particular, a formal language definition provides the basis for formal verification of programs against properties of interest.
- Modern programming languages have formal semantics to allow automated properties certification, e.g. Standard ML, Java, C, Haskell.
- New advances in programming languages towards certification of programs and properties, e.g. Microsoft F*

Approaches to formal semantics

- Denotational semantics.
- Operational semantics.
  - Big-step semantics
  - Small-step semantics
  - Structural Operational Semantics (SOS)
  - Modular SOS
  - Reduction semantics
  - Continuation-based semantics
  - Abstract State Machine (ASM)
- Axiomatic semantics.
- Action Semantics.
- Concurrency semantics.

Interpreters???

- We define the executable semantics of different programming languages
- Keep in mind that we do not provide an implementation of the language - that would be the job of an interpreter or a compiler.
- What we provide is a semantics of the language as a formal mathematical definition of the language.
- The operations and properties, forming together what is called a specification, represent the totality of properties which if an implementation satisfies, that implementation is considered correct for our language.

Outline

Denotational Semantics

Denotational Semantics

- Associates to each programming language syntactic construct a well-defined and understood mathematical object, typically a function.
- Also known as fix-point semantics
- The mathematical object denotes the behavior of the corresponding language construct, so equivalence of programs is immediately translated into equivalence of mathematical objects (bi-simulation, equivalence by abstraction, etc.)
- Undefinedness is denoted by $\perp$ and every denotational definition must propagate $\perp$:
  $$[\![a_1 + a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma + [\![a_2]\!]\sigma & \text{if } [\![a_1]\!]\sigma \neq \perp \text{ and } [\![a_2]\!]\sigma \neq \perp \\ \perp & \text{otherwise} \end{cases}$$
  instead of the simpler $[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma + [\![a_2]\!]\sigma$

Example Program in SIMPLE

```
x := 0 ;
n := 0 ;
while (n <= 2) (
  x := x + 1 ;
  n := n + x
)
```

- Imperative style (no concurrency)
- Assignment, loop and conditional
- Data: only integer numbers
- Data: boolean only in conditions

Syntax in SIMPLE

$$
\begin{array}{rl}
\mathit{Var} ::= & \text{standard identifiers} \\
\mathit{AExp} ::= & \mathit{Var} \mid 1 \mid 2 \mid 3 \mid \ldots \mid \\
& \mathit{AExp} + \mathit{AExp} \mid \mathit{AExp} - \mathit{AExp} \mid \mathit{AExp} * \mathit{AExp} \mid \mathit{AExp} / \mathit{AExp} \\
\mathit{BExp} ::= & \mathit{true} \mid \mathit{false} \mid \mathit{AExp} \leq \mathit{AExp} \mid \mathit{AExp} \geq \mathit{AExp} \mid \mathit{AExp} == \mathit{AExp} \\
& \mathit{BExp}\ \mathit{and}\ \mathit{BExp} \mid \mathit{BExp}\ \mathit{or}\ \mathit{BExp} \mid \mathit{not}\ \mathit{BExp} \\
\mathit{Stmt} ::= & \mathit{skip} \mid \mathit{Var} := \mathit{AExp} \mid \mathit{Stmt}\ ;\ \mathit{Stmt} \\
& \mathit{if}\ \mathit{BExp}\ \mathit{then}\ \mathit{Stmt}\ \mathit{else}\ \mathit{Stmt} \mid \mathit{while}\ \mathit{BExp}\ \mathit{Stmt} \\
\mathit{Pgm} ::= & \mathit{Stmt}
\end{array}
$$

State and Configuration in SIMPLE

A state is a mapping (i.e., substitution) from variables to integer numbers

$$\sigma : Var \mapsto Int$$

Retrieve a value	$\sigma[x]$
Assign a value	$\sigma[x \leftarrow i]$

A configuration is just the program and the state

$$\langle a, \sigma \rangle$$

We assume a language w/o side-effects.
Three groups of denotational evaluation

$$\begin{aligned}
[\![_]\!] &: AExp \rightarrow ((Var \mapsto Int) \rightarrow Int) \\
[\![_]\!] &: BExp \rightarrow ((Var \mapsto Int) \rightarrow Bool) \\
[\![_]\!] &: Stmt \rightarrow ((Var \mapsto Int) \rightarrow (Var \mapsto Int))
\end{aligned}$$

Denotational Semantics of SIMPLE (I)

Denotational evaluation of arithmetic expressions:

- both arguments are recursively evaluated
- functions $+$, $-$ and $*$ are extended for $\perp$ but not shown
- function $/$ has special consideration and also extended for $\perp$

$$[\![\_]\!] : AExp \to ((Var \mapsto Int) \to Int)$$

$$[\![x]\!]\sigma = \sigma(x) \qquad [\![i]\!]\sigma = i$$

$$[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma + [\![a_2]\!]\sigma \qquad [\![a_1 - a_2]\!]\sigma = [\![a_1]\!]\sigma - [\![a_2]\!]\sigma \qquad [\![a_1 * a_2]\!]\sigma = [\![a_1]\!]\sigma * [\![a_2]\!]\sigma$$

$$[\![a_1 / a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma / [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \perp & \text{otherwise} \end{cases}$$

Denotational Semantics of SIMPLE (II)

Denotational evaluation of boolean expressions:

- both arguments are recursively evaluated
- all functions are extended for $\perp$ but not shown
- functions (and), (or) denote the version of and, or used for implementation

$$[\![\_]\!] : BExp \rightarrow ((Var \mapsto Int) \rightarrow Bool)$$

$$[\![true]\!]\sigma = true \qquad [\![false]\!]\sigma = false \qquad [\![not\ \ b]\!]\sigma = \neg([\![b]\!]\sigma)$$

$$[\![b_1 == b_2]\!]\sigma = [\![b_1]\!]\sigma(==)[\![b_2]\!]\sigma$$

$$[\![b_1 \leq b_2]\!]\sigma = [\![b_1]\!]\sigma \leq [\![b_2]\!]\sigma \qquad [\![b_1 \geq b_2]\!]\sigma = [\![b_1]\!]\sigma \geq [\![b_2]\!]\sigma$$

$$[\![b_1\ \ and\ \ b_2]\!]\sigma = [\![b_1]\!]\sigma\ (and)\ [\![b_2]\!]\sigma \qquad [\![b_1\ \ or\ \ b_2]\!]\sigma = [\![b_1]\!]\sigma\ (or)\ [\![b_2]\!]\sigma$$

Denotational Semantics of SIMPLE (III)

Denotational evaluation of instructions:

- recursive use of arithmetic and boolean evaluation functions
- instructions ; and := are extended for $\perp$ but not shown
- conditional and loop instructions have special consideration (extended version)

$$[\![\_]\!] : Stmt \to ((Var \mapsto Int) \to (Var \mapsto Int))$$

$$[\![skip]\!]\sigma = \sigma$$

$$[\![x := a]\!]\sigma = \sigma[x \leftarrow [\![a]\!]\sigma]$$

$$[\![s_1 \ ; \ s_2]\!]\sigma = [\![s_2]\!]([\![s_1]\!]\sigma)$$

$$[\![if \ \ b \ then \ s_1 \ \ else \ \ s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = true \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = false \\ \perp & \text{if } [\![b]\!]\sigma = \perp \end{cases}$$

Denotational Semantics of SIMPLE (IV)

Denotational evaluation of loops:

- Obtain the fix point of an iterative version of the semantics
- The iterative semantics needs only a local version of `while` but makes recursive calls to the $[\![\_]\!]$ function

$$[\![\_]\!] : Stmt \to ((Var \mapsto Int) \to (Var \mapsto Int))$$

$$[\![\_]\!]_{while} : Stmt \to ((Var \mapsto Int) \to (Var \mapsto Int))$$

$$[\![while\ b\ s]\!]\sigma = fixpoint^{\infty}([\![while\ b\ s]\!]_{while}, \sigma)$$

$$[\![while\ b\ s]\!]_{while}\sigma = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = false \\ [\![s]\!]\sigma & \text{if } [\![b]\!]\sigma = true \\ \perp & \text{if } [\![b]\!]\sigma = \perp \end{cases}$$

$$[\![\texttt{while (true) skip}]\!] = ??? \qquad [\![\texttt{while (true) } x := x + 1]\!] = ???$$

Example of execution

$$\llbracket x := 0; n := 0; while(n <= 2)(x := x + 1; n := n + x)\rrbracket\sigma_0 \quad (\sigma_0 = \varnothing)$$
$$= \llbracket n := 0; while(n <= 2)(x := x + 1; n := n + x)\rrbracket\sigma_1 \quad (\sigma_1 = \{\mathtt{x} \mapsto 0\})$$
$$= \llbracket while(n <= 2)(x := x + 1; n := n + x)\rrbracket\sigma_2 \quad (\sigma_2 = \{\mathtt{x} \mapsto 0, \mathtt{n} \mapsto 0\})$$
$$= fixpoint^{\infty}(\llbracket while...\rrbracket_{\mathtt{while}}, \sigma_2) = ... = \sigma_6$$

$$\llbracket while(n <= 2)(x := x + 1; n := n + x)\rrbracket_{\mathtt{while}}\sigma_2$$
$$= \llbracket x := x + 1; n := n + x\rrbracket\sigma_2 \quad \text{because } \llbracket n <= 2\rrbracket\sigma_2 = true$$
$$= \llbracket n := n + x\rrbracket\sigma_3 \quad (\sigma_3 = \{\mathtt{x} \mapsto 1, \mathtt{n} \mapsto 0\})$$
$$= \sigma_4 \quad (\sigma_4 = \{\mathtt{x} \mapsto 1, \mathtt{n} \mapsto 1\})$$

$$\llbracket while(n <= 2)(x := x + 1; n := n + x)\rrbracket_{\mathtt{while}}\sigma_4$$
$$= \llbracket x := x + 1; n := n + x\rrbracket\sigma_4 \quad \text{because } \llbracket n <= 2\rrbracket\sigma_4 = true$$
$$= \llbracket n := n + x\rrbracket\sigma_5 \quad (\sigma_5 = \{\mathtt{x} \mapsto 2, \mathtt{n} \mapsto 1\})$$
$$= \sigma_6 \quad (\sigma_6 = \{\mathtt{x} \mapsto 2, \mathtt{n} \mapsto 3\})$$

$$\llbracket while(n <= 2)(x := x + 1; n := n + x)\rrbracket_{\mathtt{while}}\sigma_6 = \sigma_6$$

Exercises (I)

Which is the meaning of these programs?

1. `if true and (true or false) then x:= 1 else x:= 2`
2. `skip`
3. `x := 100 + 100`
4. `x := 1 ; if x == 1 then y := 2 else y := 3`
5. `x := 1 ; while x >= 1 x := x - 1`
6. `x := 0 ; n := 0 ; while n <= 2 (x := x+1; n := n+x)`

Exercises (II)

Which is the meaning of these programs?

- $[\![\texttt{skip}]\!]$
- $[\![1/0]\!]$
- $[\![\texttt{while (true) skip}]\!]$
- $[\![\texttt{while (true) } x := x + 1]\!]$

Outline

Operational Semantics

Operational Semantics (I)

- Collection of axioms specifying how its expressions, statements, etc., are evaluated.

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle, \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 + i_2, \sigma_2\rangle}$$

  "Operational" because they say how a possible implementation of a programming language should "operate".

- No agreement how an operational semantics of a language should be given. Any rigorous enough description is valid, Even an adhoc implementation.

Operational Semantics (II)

- Different formalisms for describing operational semantics:
  - Structural Operational Semantics
  - Chemical machine
  - Virtual abstract machine (JVM)
  - Even an adhoc implementation (SUN implementation of Java)
- Formal operational semantics if a collection of rules in some rigorous logical formalism.
  Advantage: formally reason about programs.
- Two families of operational semantics
  - Big-step operational semantics
  - Small-step operational semantics

Big-step operational semantics

- Relations on system configurations ($C \Downarrow C'$)
- Also known as natural semantics.
- How final evaluation result of each language construct can be obtained by combining the evaluation results of their syntactic counterparts.
- For example (with side effects)

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle, \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 (+) i_2, \sigma_2\rangle}$$

- For example (w/o side effects):

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle, \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 (+) i_2\rangle}$$

Small-step operational semantics

- "One-computation-step" transitions.
- Also known as transitional semantics.
- For each language construct take one of its syntactic counterparts and show that translates into one-computation step.

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma'\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma'\rangle} \quad \frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma'\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma'\rangle}$$

$$\frac{\cdot}{\langle i_1 + i_2, \sigma\rangle \rightarrow \langle i_1 (+) i_2\rangle}$$

Difference between intermediate (auxiliary) and final steps (observable).

- A big-step corresponds to many steps small-step

Outline

4. Operational Semantics
   - Big-step Operational Semantics
   - Small-step Operational Semantics

Big-step Definition of SIMPLE (I)

Big-step evaluation of arithmetic expressions:

- Two configurations $\langle \mathit{expression}, \mathit{store} \rangle$ and $\langle \mathit{value} \rangle$
- both arguments are recursively evaluated
- Here there is no consideration for $\perp$!!!!
- Function $/$ has a condition

$$\frac{\cdot}{\langle x, \sigma \rangle \Downarrow \langle \sigma(x) \rangle} \qquad \frac{\cdot}{\langle i, \sigma \rangle \Downarrow \langle i \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle, \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 (+) i_2 \rangle} \qquad \frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle, \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 - a_2, \sigma \rangle \Downarrow \langle i_1 (-) i_2 \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle, \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 * a_2, \sigma \rangle \Downarrow \langle i_1 (*) i_2 \rangle} \qquad \frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle, \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle i_1 (/) i_2 \rangle}\ i_2 \neq 0$$

Example Proof-tree in Big-step SIMPLE

$$\dfrac{\dfrac{\cdot}{\langle x, \sigma \rangle \Downarrow 1}, \quad \dfrac{\dfrac{\dfrac{\cdot}{\langle y, \sigma \rangle \Downarrow 1}, \dfrac{\cdot}{\langle x, \sigma \rangle \Downarrow 1}}{\langle y * x, \sigma \rangle \Downarrow 1}, \dfrac{\cdot}{\langle 2, \sigma \rangle \Downarrow 2}}{\langle y * x + 2, \sigma \rangle \Downarrow 3}}{\langle x - (y * x + 2), \sigma \rangle \Downarrow -2}$$

$$\sigma \equiv \{x \mapsto 1, y \mapsto 1\}$$

Big-step Definition of SIMPLE (II)

Big-step evaluation of boolean expressions:

- Two configurations $\langle expression, store \rangle$ and $\langle value \rangle$
- both arguments are recursively evaluated
- Here there is no consideration for $\perp$!!!!
- functions with parenthesis denote the version used for implementation

$$\frac{.}{\langle true, \sigma \rangle \Downarrow \langle true \rangle} \qquad \frac{.}{\langle false, \sigma \rangle \Downarrow \langle false \rangle}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle b \rangle}{\langle not\ \ e, \sigma \rangle \Downarrow \langle \neg b \rangle} \qquad \frac{\langle e_1, \sigma \rangle \Downarrow \langle b_1 \rangle, \langle e_2, \sigma \rangle \Downarrow \langle b_2 \rangle}{\langle e_1\ \ ==\ \ e_2, \sigma \rangle \Downarrow \langle b_1 (==) b_2 \rangle}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow \langle b_1 \rangle, \langle e_2, \sigma \rangle \Downarrow \langle b_2 \rangle}{\langle e_1\ \ and\ \ e_2, \sigma \rangle \Downarrow \langle b_1 (and) b_2 \rangle} \qquad \frac{\langle e_1, \sigma \rangle \Downarrow \langle b_1 \rangle, \langle e_2, \sigma \rangle \Downarrow \langle b_2 \rangle}{\langle e_1\ \ or\ \ e_2, \sigma \rangle \Downarrow \langle b_1 (or) b_2 \rangle}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow \langle b_1 \rangle, \langle e_2, \sigma \rangle \Downarrow \langle b_2 \rangle}{\langle e_1 \leq e_2, \sigma \rangle \Downarrow \langle b_1 (\leq) b_2 \rangle} \qquad \frac{\langle e_1, \sigma \rangle \Downarrow \langle b_1 \rangle, \langle e_2, \sigma \rangle \Downarrow \langle b_2 \rangle}{\langle e_1 \geq e_2, \sigma \rangle \Downarrow \langle b_1 (\geq) b_2 \rangle}$$

Big-step Definition of SIMPLE (III)

Big-step evaluation of instructions:

- Two configurations $\langle expression, store\rangle$ and $\langle store\rangle$
- Here there is no consideration for $\perp$!!!!
- Here there are no different evaluations for arithmetic and boolean
- Big-step without side effects
- no special treatment for conditional or loop instructions

$$\frac{\cdot}{\langle skip, \sigma\rangle \Downarrow \langle\sigma\rangle} \qquad \frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle}{\langle x := a, \sigma\rangle \Downarrow \langle\sigma[x \leftarrow i]\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle\sigma_1\rangle, \langle s_2, \sigma_1\rangle \Downarrow \langle\sigma_2\rangle}{\langle s_1\ ;\ s_2, \sigma\rangle \Downarrow \langle\sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle true\rangle, \langle s_1, \sigma\rangle \Downarrow \langle\sigma'\rangle}{\langle if\ \ e_1\ \ then\ s_1\ \ else\ \ s_2, \sigma\rangle \Downarrow \langle\sigma'\rangle} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow \langle false\rangle, \langle s_2, \sigma\rangle \Downarrow \langle\sigma'\rangle}{\langle if\ \ e_1\ \ then\ s_1\ \ else\ \ s_2, \sigma\rangle \Downarrow \langle\sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle true\rangle, \langle s, \sigma\rangle \Downarrow \langle\sigma'\rangle, \langle while\ \ e\ \ s, \sigma'\rangle \Downarrow \langle\sigma''\rangle}{\langle while\ \ e\ \ s, \sigma\rangle \Downarrow \langle\sigma''\rangle} \qquad \frac{\langle e, \sigma\rangle \Downarrow \langle false\rangle}{\langle while\ \ e\ \ s, \sigma\rangle \Downarrow \langle\sigma\rangle}$$

Example Proof-tree in Big-step SIMPLE

$$
\frac{
  \dfrac{\cdot}{\langle x, \sigma_0\rangle \Downarrow \sigma_1},
  \quad
  \dfrac{
    \dfrac{\cdot}{\langle n, \sigma_1\rangle \Downarrow \sigma_2},
    \dfrac{
      \dfrac{\vdots}{\langle n <= 2, \sigma_3\rangle \Downarrow \langle \mathit{true}\rangle},
      \dfrac{\vdots}{\cdots},
      \dfrac{\vdots}{\cdots}
    }{\langle \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_0\rangle \Downarrow \sigma_{k-2}}
  }{\langle n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_0\rangle \Downarrow \sigma_{k-1}}
}{\langle x := 0; n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_0\rangle \Downarrow \sigma_k}
$$

$$\sigma_0 \equiv \varnothing,\ \ \sigma_1 \equiv \{x \mapsto 0\},\ \ \sigma_2 \equiv \{x \mapsto 0, n \mapsto 0\},\ \ \sigma_2 \equiv \sigma_3, \ldots, \sigma_k \equiv \{x \mapsto 2, n \mapsto 3\}$$

Exercises (I)

Which is the meaning of these programs?

1. `if true and (true or false) then x:= 1 else x:= 2`
2. `skip`
3. `x := 100 + 100`
4. `x := 1 ; if x == 1 then y := 2 else y := 3`
5. `x := 1 ; while x >= 1 x := x - 1`
6. `x := 0 ; n := 0 ; while n <= 2 (x := x+1; n := n+x)`

Exercises (II)

Which is the meaning of these programs?

- $[\![\mathtt{skip}]\!]$
- $[\![1/0]\!]$
- $[\![\mathtt{while}\ (\mathtt{true})\ \mathtt{skip}]\!]$
- $[\![\mathtt{while}\ (\mathtt{true})\ x := x + 1]\!]$

Outline

Small-step Definition of SIMPLE (I)

Small-step evaluation of arithmetic expressions:

- Two configurations $\langle expression, store \rangle$ and $\langle value \rangle$
- both arguments shall be recursively evaluated???
- Function $/$ has a condition

$$\frac{\cdot}{\langle x, \sigma \rangle \to \langle \sigma(x) \rangle} \qquad \frac{\cdot}{\langle i, \sigma \rangle \to \langle i \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \to \langle a_1' + a_2, \sigma \rangle} \quad \frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \to \langle a_1 + a_2', \sigma \rangle} \quad \frac{\cdot}{\langle i_1 + i_2, \sigma \rangle \to \langle i_1 (+) i_2 \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 - a_2, \sigma \rangle \to \langle a_1' - a_2, \sigma \rangle} \quad \frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle a_1 - a_2, \sigma \rangle \to \langle a_1 - a_2', \sigma \rangle} \quad \frac{\cdot}{\langle i_1 - i_2, \sigma \rangle \to \langle i_1 (-) i_2 \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 * a_2, \sigma \rangle \to \langle a_1' * a_2, \sigma \rangle} \quad \frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle a_1 * a_2, \sigma \rangle \to \langle a_1 * a_2', \sigma \rangle} \quad \frac{\cdot}{\langle i_1 * i_2, \sigma \rangle \to \langle i_1 (*) i_2 \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 / a_2, \sigma \rangle \to \langle a_1' / a_2, \sigma \rangle} \quad \frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle a_1 / a_2, \sigma \rangle \to \langle a_1 / a_2', \sigma \rangle} \quad \frac{\cdot}{\langle i_1 / i_2, \sigma \rangle \to \langle i_1 (/) i_2 \rangle} \; i_2 \neq 0$$

Example Proof-tree in Small-step SIMPLE

$$\langle x - (y * x + 2), \sigma\rangle \rightarrow \langle 1 - (y * x + 2), \sigma\rangle \rightarrow \langle 1 - (1 * x + 2), \sigma\rangle$$
$$\rightarrow \langle 1 - (1 * 1 + 2), \sigma\rangle \rightarrow \langle 1 - (1 + 2), \sigma\rangle \rightarrow \langle 1 - 3, \sigma\rangle \rightarrow \langle -2, \sigma\rangle \rightarrow \langle -2\rangle$$

$$\sigma \equiv \{x \mapsto 1, y \mapsto 1\}$$

Small-step Definition of SIMPLE (II)

Small-step evaluation of boolean expressions:

- Two configurations $\langle expression, store\rangle$ and $\langle value\rangle$
- both arguments shall be recursively evaluated??
- functions with parenthesis denote the version used for implementation

$$\frac{\cdot}{\langle true, \sigma\rangle \rightarrow \langle true\rangle} \qquad \frac{\cdot}{\langle false, \sigma\rangle \rightarrow \langle false\rangle}$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle e', \sigma\rangle}{\langle not\ e, \sigma\rangle \rightarrow \langle not\ e', \sigma\rangle} \qquad \frac{\langle e, \sigma\rangle \rightarrow \langle b\rangle}{\langle not\ e, \sigma\rangle \rightarrow \langle \neg b\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 == a_2, \sigma\rangle \rightarrow \langle a_1' == a_2, \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 == a_2, \sigma\rangle \rightarrow \langle a_1 == a_2', \sigma\rangle} \qquad \frac{\cdot}{\langle i_1 == i_2, \sigma\rangle \rightarrow \langle i_1(==)i_2\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1\ and\ a_2, \sigma\rangle \rightarrow \langle a_1'\ and\ a_2, \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1\ and\ a_2, \sigma\rangle \rightarrow \langle a_1\ and\ a_2', \sigma\rangle} \qquad \frac{\cdot}{\langle i_1\ and\ i_2, \sigma\rangle \rightarrow \langle i_1(and)i_2\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1\ or\ a_2, \sigma\rangle \rightarrow \langle a_1'\ or\ a_2, \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1\ or\ a_2, \sigma\rangle \rightarrow \langle a_1\ or\ a_2', \sigma\rangle} \qquad \frac{\cdot}{\langle i_1\ or\ i_2, \sigma\rangle \rightarrow \langle i_1(or)i_2\rangle}$$

Small-step Definition of SIMPLE (III)

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \leq a_2, \sigma\rangle \to \langle a_1' \leq a_2, \sigma\rangle} \quad \frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \leq a_2, \sigma\rangle \to \langle a_1 \leq a_2', \sigma\rangle} \quad \frac{\cdot}{\langle i_1 \leq i_2, \sigma\rangle \to \langle i_1(\leq)i_2\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \geq a_2, \sigma\rangle \to \langle a_1' \geq a_2, \sigma\rangle} \quad \frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \geq a_2, \sigma\rangle \to \langle a_1 \geq a_2', \sigma\rangle} \quad \frac{\cdot}{\langle i_1 \geq i_2, \sigma\rangle \to \langle i_1(\geq)i_2\rangle}$$

Note is not an error σ instead of σ' in $\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle$

Small-step Definition of SIMPLE (IV)

Small-step evaluation of instructions:

- Small-step without side effects but easy to change
- Two configurations $\langle \mathit{expression}, \mathit{store}\rangle$ and $\langle \mathit{value}\rangle$
- no special treatment for conditional or loop instructions
- loops are syntactic sugar for conditional instructions

$$\frac{\cdot}{\langle \mathit{skip}, \sigma\rangle \to \langle\sigma\rangle} \qquad \frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle x := a, \sigma\rangle \to \langle x := a', \sigma\rangle} \qquad \frac{\cdot}{\langle x := i, \sigma\rangle \to \langle \sigma[x \leftarrow i]\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle}{\langle s_1 \ ; \ s_2, \sigma\rangle \to \langle s_1' \ ; \ s_2, \sigma'\rangle} \qquad \frac{\langle s_1, \sigma\rangle \to \langle \sigma'\rangle}{\langle s_1 \ ; \ s_2, \sigma\rangle \to \langle s_2, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to \langle e', \sigma\rangle}{\langle \mathit{if}\ \ e\ \ \mathit{then}\ s_1\ \ \mathit{else}\ s_2, \sigma\rangle \to \langle \mathit{if}\ \ e'\ \ \mathit{then}\ s_1\ \ \mathit{else}\ s_2, \sigma\rangle}$$

$$\frac{\cdot}{\langle \mathit{if}\ \ \mathit{true}\ \ \mathit{then}\ s_1\ \ \mathit{else}\ s_2, \sigma\rangle \to \langle s_1, \sigma\rangle} \qquad \frac{\cdot}{\langle \mathit{if}\ \ \mathit{false}\ \ \mathit{then}\ s_1\ \ \mathit{else}\ s_2, \sigma\rangle \to \langle s_2, \sigma\rangle}$$

$$\frac{\cdot}{\langle \mathit{while}\ \ e\ \ s, \sigma\rangle \to \langle \mathit{if}\ \ e\ \ \mathit{then}\ (s\ ;\ \ \mathit{while}\ \ e\ \ s)\ \ \mathit{else}\ \mathit{skip}, \sigma\rangle}$$

Example

$$
\begin{aligned}
&\langle x := 0; n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_0 \rangle \qquad (\sigma_0 = \varnothing)\\
&\rightarrow \langle n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_1 \rangle \qquad (\sigma_1 = \{x \mapsto 0\})\\
&\rightarrow \langle \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_1 \rangle \qquad (\sigma_1 = \{x \mapsto 0, n \mapsto 0\})\\
&\rightarrow \langle \mathit{if}(n <= 2)\ \mathit{then}\ ((x := x + 1; n := n + x); \mathit{while}\ldots)\ \mathit{else}\ \mathit{skip}, \sigma_1 \rangle\\
&\rightarrow \langle \mathit{if}(0 <= 2)\ \mathit{then}\ ((x := x + 1; n := n + x); \mathit{while}\ldots)\ \mathit{else}\ \mathit{skip}, \sigma_1 \rangle\\
&\rightarrow \langle \mathit{if}(\mathit{true})\ \mathit{then}\ ((x := x + 1; n := n + x); \mathit{while}\ldots)\ \mathit{else}\ \mathit{skip}, \sigma_1 \rangle\\
&\rightarrow \langle x := x + 1; n := n + x; \mathit{while}\ldots, \sigma_1 \rangle\\
&\rightarrow \langle x := 0 + 1; n := n + x; \mathit{while}\ldots, \sigma_1 \rangle\\
&\rightarrow \langle x := 1; n := n + x; \mathit{while}\ldots, \sigma_1 \rangle\\
&\rightarrow \langle n := n + x; \mathit{while}\ldots, \sigma_2 \rangle \qquad (\sigma_2 = \{x \mapsto 1, n \mapsto 0\})\\
&\ \vdots\\
&\rightarrow \langle \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_3 \rangle \qquad (\sigma_3 = \{x \mapsto 1, n \mapsto 1\})\\
&\ \vdots\\
&\rightarrow \langle \mathit{while}(n <= 2)(x := x + 1; n := n + x), \sigma_4 \rangle \qquad (\sigma_4 = \{x \mapsto 2, n \mapsto 3\})\\
&\ \vdots\\
&\rightarrow \langle \mathit{skip}, \sigma_4 \rangle\\
&\rightarrow \langle \sigma_4 \rangle
\end{aligned}
$$

Exercises (I)

Which is the meaning of these programs?

1. `if true and (true or false) then x:= 1 else x:= 2`
2. `skip`
3. `x := 100 + 100`
4. `x := 1 ; if x == 1 then y := 2 else y := 3`
5. `x := 1 ; while x >= 1 x := x - 1`
6. `x := 0 ; n := 0 ; while n <= 2 (x := x+1; n := n+x)`

Exercises (II)

Which is the meaning of these programs?

- $[\![\texttt{skip}]\!]$
- $[\![1/0]\!]$
- $[\![\texttt{while (true) skip}]\!]$
- $[\![\texttt{while (true) } x := x + 1]\!]$

Big-step vs Small-step

- Advantages of Big-Step
  - When it can be given to a language, it is easier to understand because it relates syntactic entities directly to their expected results
  - More abstract, more mathematical; therefore, one can more easily define and prove properties about programs
- Disadvantages of Big-Step
  - It is not executable. One always needs to provide a result value or state to each language construct and then use the inference rules to "check" whether that result value or state is indeed correct.
  - Hides non-termination of programs. Non-termination as "lack of proof" (as a division by zero or runtime error).
  - Inconvenient (impossible) for non-deterministic or parallel languages.

Big-step vs Small-step

- Advantages of Small-Step
  - It is executable. Each successful applications of small-step rules denotes a real execution step; also helps locating problems or errors in programs/semantics.
  - Non-termination of programs equals non-termination of searching for a proof. But division by zero (or runtime errors) can still be located.
  - Non-deterministic and/or parallel languages.
- Disadvantages of Small-Step
  - Less suitable for proving properties about programs; use correspondence big-step & small-step.
  - Too low level and explicit; too many language definitions.
  - It has a rigid computation granularity.

Big-step vs Small-step: Disadvantages of both

- Not modular; adding new language features.
- If one adds features with (abruptly) values (exceptions, break/continue in loops, threads, etc.) then one needs extra data stores in configurations (all definitions must change)
- No appropriate semantical foundation for concurrent languages. Big-step no meaningful concurrent language Small-step only interleaving semantics.
- They are both operational and syntax-driven, so they tell us close to nothing about models of languages.

Outline

Continuation Style

- Continuation Passing Style (CPS) is a framework for enconding lambda-calculus expressions
- All elements, features, intermediate results, and final results are wrapped by corresponding continuation operations $\curvearrowright$
- At any moment during the execution, the continuation structure contains everything needed to continue the execution of the program:

$$\mathrm{HALT} \;\Rightarrow\; a_1 + a_2 \curvearrowright \mathrm{HALT} \;\Rightarrow\; a_1 \curvearrowright \square + a_2 \curvearrowright \mathrm{HALT} \;\Rightarrow\; \cdots$$

  One can simply stop the evaluation, save its state as a "core dump", and restart later
- The state contains "everything needed" to continue the execution of the program till the end.
- The configuration of the "executing" program contains several attributes, including the continuation and the store σ

Continuation-Style Definition of SIMPLE (I#)

Continuation evaluation of arithmetic expressions:

- both arguments shall be recursively evaluated in parallel???

$$a_1 + a_2 \curvearrowright K \leftrightharpoons a_1 \curvearrowright \square + a_2 \curvearrowright K \qquad a_1 + a_2 \curvearrowright K \leftrightharpoons a_2 \curvearrowright a_1 + \square \curvearrowright K$$
$$\langle i_1 \rangle \curvearrowright \square + a_2 \curvearrowright K \leftrightharpoons a_2 \curvearrowright i_1 + \square \curvearrowright K \qquad \langle i_2 \rangle \curvearrowright a_1 + \square \curvearrowright K \leftrightharpoons a_1 \curvearrowright \square + i_2 \curvearrowright K$$
$$i_1 + i_2 \curvearrowright K \leftrightharpoons \langle i_1 + i_2 \rangle \curvearrowright K$$

Continuation-Style Definition of SIMPLE (I)

Continuation evaluation of arithmetic expressions:

- Always one configuration: *continuation* | *store* but sometimes store is omitted
- Two kinds of continuations *expression* and $\langle$*value*$\rangle$
- Reduce semantics operations and include more annotations
- function / has a condition

$$x \curvearrowright K \mid \sigma \leftrightharpoons \langle \sigma[x] \rangle \curvearrowright K \mid \sigma \qquad\qquad i \curvearrowright K \leftrightharpoons \langle i \rangle \curvearrowright K$$

$$\begin{array}{rl}
a_1 + a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square + \square \curvearrowright K & \langle i_1, i_2 \rangle \curvearrowright \square + \square \curvearrowright K \leftrightharpoons \langle i_1 + i_2 \rangle \curvearrowright K \\
a_1 - a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square - \square \curvearrowright K & \langle i_1, i_2 \rangle \curvearrowright \square - \square \curvearrowright K \leftrightharpoons \langle i_1 - i_2 \rangle \curvearrowright K \\
a_1 * a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square * \square \curvearrowright K & \langle i_1, i_2 \rangle \curvearrowright \square * \square \curvearrowright K \leftrightharpoons \langle i_1 * i_2 \rangle \curvearrowright K \\
a_1 / a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square / \square \curvearrowright K & \langle i_1, i_2 \rangle \curvearrowright \square / \square \curvearrowright K \leftrightharpoons \langle i_1 / i_2 \rangle \curvearrowright K \text{ IF } i_2 \neq 0
\end{array}$$

Continuation-Style Definition of SIMPLE (II)

Continuation evaluation of boolean expressions:

- Always one configuration: *continuation* | *store* but sometimes store is omitted
- Two kinds of continuations *expression* and $\langle$*value*$\rangle$

$$
\begin{array}{rr}
\mathit{true} \curvearrowright K \leftrightharpoons \langle \mathit{true} \rangle \curvearrowright K & \mathit{false} \curvearrowright K \leftrightharpoons \langle \mathit{false} \rangle \curvearrowright K \\
\mathit{not}\ e \curvearrowright K \leftrightharpoons e \curvearrowright \mathit{not}\ \square \curvearrowright K & \langle b \rangle \curvearrowright \mathit{not}\ \square \curvearrowright K \leftrightharpoons \langle \mathit{not}\ b \rangle \curvearrowright K \\
a_1 == a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square == \square \curvearrowright K & \langle b_1, b_2 \rangle \curvearrowright \square == \square \curvearrowright K \leftrightharpoons \langle b_1 == b_2 \rangle \curvearrowright K \\
a_1\ \mathit{and}\ a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square\ \mathit{and}\ \square \curvearrowright K & \langle b_1, b_2 \rangle \curvearrowright \square\ \mathit{and}\ \square \curvearrowright K \leftrightharpoons \langle b_1\ \mathit{and}\ b_2 \rangle \curvearrowright K \\
a_1\ \mathit{or}\ a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square\ \mathit{or}\ \square \curvearrowright K & \langle b_1, b_2 \rangle \curvearrowright \square\ \mathit{or}\ \square \curvearrowright K \leftrightharpoons \langle b_1\ \mathit{or}\ b_2 \rangle \curvearrowright K \\
\\
a_1 \leq a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square \leq \square \curvearrowright K & \langle b_1, b_2 \rangle \curvearrowright \square \leq \square \curvearrowright K \leftrightharpoons \langle b_1 \leq b_2 \rangle \curvearrowright K \\
a_1 \geq a_2 \curvearrowright K \leftrightharpoons (a_1, a_2) \curvearrowright \square \geq \square \curvearrowright K & \langle b_1, b_2 \rangle \curvearrowright \square \geq \square \curvearrowright K \leftrightharpoons \langle b_1 \geq b_2 \rangle \curvearrowright K
\end{array}
$$

Continuation-Style Definition of SIMPLE (III)

Continuation evaluation of instructions:

- Always one configuration: *continuation* | *store* but sometimes store is omitted
- Two kinds of continuations *expression* and $\langle value \rangle$

$$skip \curvearrowright K \leftrightharpoons K$$

$$x := a \curvearrowright K \leftrightharpoons a \curvearrowright x := \square \curvearrowright K \qquad \langle i \rangle \curvearrowright x := \square \curvearrowright K \mid \sigma \leftrightharpoons K \mid \sigma[x \leftarrow i]$$

$$s_1 \ ; \ s_2 \curvearrowright K \leftrightharpoons s_1 \curvearrowright s_2 \curvearrowright K$$

$$if \ \ e \ \ then \ \ s_1 \ \ else \ \ s_2 \curvearrowright K \leftrightharpoons e \curvearrowright if \ \square \ then \ \ s_1 \ \ else \ \ s_2 \curvearrowright K$$

$$\langle true \rangle \curvearrowright if \ \square \ then \ \ s_1 \ \ else \ \ s_2 \curvearrowright K \leftrightharpoons s_1 \curvearrowright K \qquad \langle false \rangle \curvearrowright if \ \square \ then \ \ s_1 \ \ else \ \ s_2 \curvearrowright K \leftrightharpoons s_2 \curvearrowright K$$

$$while \ \ e \ \ s \curvearrowright K \leftrightharpoons if \ \ e \ \ then \ (s \ ; \ \ while \ \ e \ \ s) \ else \ skip \curvearrowright K$$

Example

$$
\begin{aligned}
& x := 0; n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_0 \qquad (\sigma_0 = \varnothing) \\
& \leftrightharpoons x := 0 \curvearrowright n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_0 \\
& \leftrightharpoons 0 \curvearrowright x := \square \curvearrowright n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_0 \\
& \leftrightharpoons n := 0; \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_1 \qquad (\sigma_1 = \{x \mapsto 0\}) \\
& \leftrightharpoons n := 0 \curvearrowright \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_1 \\
& \leftrightharpoons 0 \curvearrowright n := \square \curvearrowright \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_1 \\
& \leftrightharpoons \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_2 \qquad (\sigma_2 = \{x \mapsto 0, n \mapsto 0\}) \\
& \leftrightharpoons \mathit{if}(n <= 2) \ \mathit{then} \ (x := x + 1; n := n + x; \mathit{while}\ldots) \ \mathit{else} \ \mathit{skip} \mid \sigma_2 \\
& \leftrightharpoons n <= 2 \curvearrowright \mathit{if} \ \square \ \mathit{then} \ (x := x + 1; n := n + x; \mathit{while}\ldots) \ \mathit{else} \ \mathit{skip} \mid \sigma_2 \\
& \leftrightharpoons \langle n, 2 \rangle \curvearrowright (\square <= \square) \curvearrowright \mathit{if} \ \square \ \mathit{then} \ (x := x + 1; n := n + x; \mathit{while}\ldots) \ \mathit{else} \ \mathit{skip} \mid \sigma_2 \\
& \leftrightharpoons \langle 0, 2 \rangle \curvearrowright (\square <= \square) \curvearrowright \mathit{if} \ \square \ \mathit{then} \ (x := x + 1; n := n + x; \mathit{while}\ldots) \ \mathit{else} \ \mathit{skip} \mid \sigma_2 \\
& \leftrightharpoons \langle 0 <= 2 \rangle \curvearrowright \mathit{if} \ \square \ \mathit{then} \ (x := x + 1; n := n + x; \mathit{while}\ldots) \ \mathit{else} \ \mathit{skip} \mid \sigma_2 \\
& \leftrightharpoons \langle \mathit{true} \rangle \curvearrowright \mathit{if} \ \square \ \mathit{then} \ (x := x + 1; n := n + x; \mathit{while}\ldots) \ \mathit{else} \ \mathit{skip} \mid \sigma_2 \\
& \leftrightharpoons x := x + 1; n := n + x; \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_2 \\
& \vdots \\
& \leftrightharpoons \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_3 \qquad (\sigma_3 = \{x \mapsto 1, n \mapsto 1\} \\
& \vdots \\
& \leftrightharpoons \mathit{while}(n <= 2)(x := x + 1; n := n + x) \mid \sigma_4 \qquad (\sigma_4 = \{x \mapsto 2, n \mapsto 3\} \\
& \vdots \\
& \leftrightharpoons \mathit{skip} \mid \sigma_4 \\
& \leftrightharpoons \sigma_4
\end{aligned}
$$

Continuation Semantics Available

- We have implemented the continuation semantics in the programming language Maude
- The implementation is available at the course repository
- Several programs can be executed and the store is returned
  1. `if true and (true or false) then x:= 1 else x:= 2`
  2. `skip`
  3. `x := 100 + 100`
  4. `x := 1 ; if x == 1 then y := 2 else y := 3`
  5. `x := 1 ; while x >= 1 x := x - 1`
  6. `x := 0 ; n := 0 ; while n <= 2 (x := x + 1; n := n + x)`

```
          \||||||||||||||||||/
        --- Welcome to Maude ---
          /||||||||||||||||||\
   Maude alpha104 built: Sep 26 2014 18:51:40
   Copyright 1997-2014 SRI International
          Wed Oct  8 18:35:53 2014
Stmt: x := 0 ; n := 0 ; while n <= 2 (x := x + 1 ; n := n + x)
Advisory: redefining module CONFIGURATION.
==========================================
rewrite in EVAL : < if true and (true or false) then x := 1 else x := 2 > .
rewrites: 32 in 0ms cpu (0ms real) (380952 rewrites/second)
result Store: [x,1]
==========================================
rewrite in EVAL : < skip > .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Store: empty
==========================================
rewrite in EVAL : < x := 100 + 100 > .
rewrites: 15 in 0ms cpu (0ms real) (~ rewrites/second)
result Store: [x,200]
==========================================
rewrite in EVAL : < x := 1 ; if x == 1 then y := 2 else y := 3 > .
rewrites: 26 in 0ms cpu (0ms real) (~ rewrites/second)
result Store: [x,1] [y,2]
==========================================
rewrite in EVAL : < x := 1 ; while x >= 1 x := x - 1 > .
rewrites: 40 in 0ms cpu (0ms real) (~ rewrites/second)
result Store: [x,0]
==========================================
rewrite in EVAL : < x := 0 ; n := 0 ; while n <= 2 (x := x + 1 ; n := n + x) > .
```