Semantics of Programming Languages

version 0.98β (January 31, 2003)

Tobias Nipkow

Winter Term 2001/2002

Contents

1 Introduction 3
- 1.1 Aims of the lecture series 3
- 1.2 The **WHILE** language 5

2 Operational Semantics 7
- 2.1 Big-step semantics 7
- 2.2 Rule induction - an introduction 9
- 2.3 Structural operational semantics 12
- 2.4 Extensions of the **WHILE** language 15
  - 2.4.1 Nondeterminism 15
  - 2.4.2 Parallelism 16
  - 2.4.3 Blocks (Local variables) 17
  - 2.4.4 Procedures 17
- 2.5 A compiler for **WHILE** 20

3 Types 24
- 3.1 Types for **WHILE** 24
- 3.2 A monomorphic type system 25
  - 3.2.1 A typesystem for $\mathbf{WHILE_T}$ 25
  - 3.2.2 Semantics for $\mathbf{WHILE_T}$ 26
  - 3.2.3 $\mathbf{WHILE_T}$ is type safe 27
- 3.3 A polymorphic type system 28
  - 3.3.1 Type checking 28
  - 3.3.2 Type inference / reconstruction 29

4 Denotational Semantics 31
- 4.1 Inductive Definitons 31
  - 4.1.1 Rule induction 31
  - 4.1.2 I_R as a least fixed point $(f(x) = x)$ 32
- 4.2 Denotational Semantics of **WHILE** 34
- 4.3 Complete partial orders (cpos) 37
  - 4.3.1 Discrete cpos 40
  - 4.3.2 Cartesian Products 41
  - 4.3.3 Function spaces 42
  - 4.3.4 Lifting 42
- 4.4 A cpo-based denotational semantics for **WHILE** 44
- 4.5 Extensions of **WHILE** 45
  - 4.5.1 Local vars and procedures 45
  - 4.5.2 Continuations and Exceptions 46
  - 4.5.3 The Knaster-Tarski Fixpoint theorem 46

4.5.4 Nondeterminism 49

5 Axiomatic Semantics (Hoare-Logic) 51
5.1 Assertions 55
5.2 The semantic approach 55
5.3 Soundness 56
5.4 Completeness 56
5.5 Expressiveness 57
5.6 Relative Completeness 59
5.7 Verification condition generation 60
5.8 Total Correctness 62
5.9 Extension of **WHILE** 63
5.9.1 Nondeterminism 63
5.9.2 Arrays 64
5.9.3 Procedures 65
5.10 A Hoare Logic for Time 69
5.10.1 Operational Semantics with time 69
5.10.2 Timed Hoare triples 70
5.10.3 Rules of Hoare Logic for Time 70

LaTeXed by Michael Wahler (wahler@in.tum.de). Please report any errors or your suggestions to me. This text is still under development and has not yet been approved by Prof. Nipkow. Thanks to Konstantinos Panagiotou for your help.

10/17/2001

1 Introduction

1.1 Aims of the lecture series

Descriptive: how to describe semantics
Analytic: how to use semantics to analyze programming language
how to analyze tools: is a compiler correct?
how to analyze programs

$$\begin{array}{rcl} c & ::= & \mathbf{skip} \\ & & \mid x := a \\ & & \mid c;\ c \\ & & \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \\ & & \mid \mathbf{while}\ b\ \mathbf{do}\ c \end{array}$$

Overview

1. **Operational Semantics**
   - (command, start state) $\rightarrow$ final state
   - inference rules $\dfrac{B(b)\sigma = true \quad (c,\sigma) \rightarrow \sigma' \quad (\mathbf{while}\ b\ \mathbf{do}\ c, \sigma') \rightarrow \sigma''}{(\mathbf{while}\ b\ \mathbf{do}\ c, \sigma) \rightarrow \sigma''}$

     B evaluation function
   - Type System, Compiler } correctness
2. **Denotational Semantics**
   - S: Syntax $\rightarrow$ Semantics
   - command $\rightarrow$(state $\nrightarrow$ state)

     $\nrightarrow$ partial function
   - $S(\mathbf{while}\ b\ \mathbf{do}\ c)\sigma = \begin{cases} \sigma \text{ if } B(b)\sigma = false \\ S(\mathbf{while}\ b\ \mathbf{do}\ c)(S(c)\sigma) \text{ otherwise} \end{cases}$
3. **Axiomatic Semantics (Hoare logic)**
   - {precondition} command {postcondition}
   - Example: $\{x := 0\} x := x+1 \{x = 2\}$ not correct!
   - correctness and completeness
   - application: runtime analysis of programs

1. $function\ f() :\ integer\ \{\ldots\};$
   $e : \mathbf{if}\ f() = f()\ \mathbf{then}\ 0\ \mathbf{else}\ 1$

   Q: how can e evaluate to 1?

   $$\{x := x+1;\ return\ x\}$$

2. $function\ f() :\ integer\ \{return f()\}$

   Q: Find e so that $e + f()$ and $f() + e$ differ.
   A: $e := \frac{1}{0}$

3.

$$\begin{array}{rl} x := 0 & \{x = 0\} \\ a[1] := 0 & \{a[1] = 0\} \\ a[a[1]] := 0 & \{a[a[1]] = 0\} \\ \{a[1] = 1 \wedge a[0] = 2\}\ a[a[1]] := 0 & \{a[a[1]] = 0\} \end{array}$$

$$\left.\begin{array}{l} addr(x) \\ addr(a[1]) \end{array}\right\} \quad \text{constant}$$

$$addr(a[a[1]]) \quad \text{can change!}$$

but works with $\{a[1] \neq 1\}$ as precondition

Q: Find context that distinguishes

$$x := 0;\ y := 1$$
and
$$y := 1;\ x := 0$$

var $z : integer$
proc $p($**var** $x, y : integer)\ \{\ldots\}$
$p(z, z)$
proc $p\ \{\ldots\}$
proc q $\{$**var** $x : int;\ x := 0;\ p\}$

Q: Can p and q be distinguished?
A: Possible Out-of-Memory-exception when calling q. p could access the stack.

proc $p($**proc** $r)$ $\{\ldots\}$ no side effects
proc q $\{$ **var** $x : int;$
proc $r\ \{x := x + 2\};$
$x := 0;\ p(r)$
if $even(x)$ **then** $\langle s\rangle\}$

Q: Is q equivalent to $\langle s\rangle$?
A: Yes, if p terminates or if $\langle s\rangle$ does not terminate.

Find context that distinguishes $1+2$ and $2+1$:

$$1+2*3 \neq 2+1*3$$
$$"1+2" \neq "2+1"$$

$A \Rightarrow B \equiv \neg A \vee B$
$A \Rightarrow (B \Rightarrow C) \equiv B \Rightarrow (A \Rightarrow C) \neq (A \Rightarrow B) \Rightarrow C$

10/22/2001

1.2 The WHILE language

Set	Meta-Var
$\mathbb{Z}$	i, j, m, n (constants)
var	x, y, z
$T = \{\text{tt}, \text{ff}\}$	

Syntax

Arithmetic expressions (*Aexp*)

$$a ::= n \mid x \mid a_1 \; aop \; a_2$$

where $aop = \{+, -, *\}$

Boolean expressions (*Bexp*)

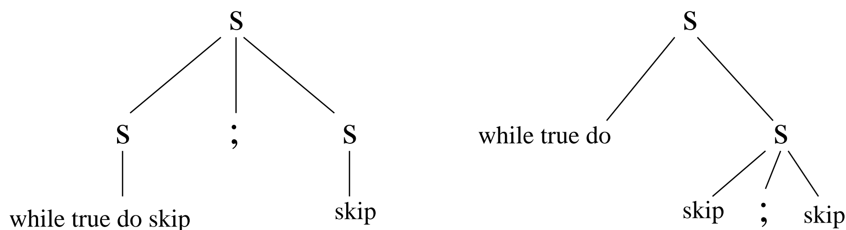
$$b ::= \textbf{true} \mid \textbf{false} \mid a \; rop \; a \mid b \; bop \; b \mid \neg b$$

where $rop = \{=, \leq\}$ and $bop = \{\wedge, \vee, \Rightarrow, \ldots\}$

Statements (*Stm*)[1]

$$s ::= \textbf{skip} \mid x := a \mid s; \; s \mid \textbf{if } b \textbf{ then } s \textbf{ else } s \mid \textbf{while } b \textbf{ do } s$$

Concrete syntax: grammars for strings, should be *unambiguous*
Stm is ambiguous: **while true do skip** ; **skip**



Abstract syntax: grammar for trees
Use (,) do disambiguate:

(**while true do skip**); **skip**

while true do (**skip** ; **skip**)

Compare tree level and string level of "2+1*3" example

The State

$$\begin{aligned} \text{State} &= \text{mapping from } \textbf{var} \text{ to } \mathbb{Z} \\ \Sigma &= \textbf{var} \rightarrow \mathbb{Z} \text{ (Meta-var: } \sigma) \end{aligned}$$

Updating the state

$$\begin{aligned} \sigma[x \mapsto n] &= \text{"}\sigma \text{ with } x \text{ bound to } n\text{"}^{2} \\ (\sigma[x \mapsto n])(y) &= \begin{cases} n & \text{if } x = y \\ \sigma(x) & \text{if } x \neq y \end{cases} \end{aligned}$$

[1] [WINSKEL93]: "command"

Concrete states: $\{x \mapsto 5, y \mapsto 7\}$

Semantics of expressions

$\mathcal{A} : Aexp \to (\Sigma \to \mathbb{Z}) \quad (Aexp \times \Sigma \to \mathbb{Z})$
$\mathcal{B} : Bexp \to \Sigma \to T$

$$\begin{array}{rcl} \mathcal{A}[n]\sigma & = & n \text{ (because } n \text{ is constant!)} \\ \mathcal{A}[x]\sigma & = & \sigma(x) \\ \mathcal{A}[a_1\ op\ a_2]\sigma & = & [aop](\mathcal{A}[a_1]\sigma, \mathcal{A}[a_2]\sigma) \end{array}$$

For each other operator $\otimes \in aop$, $[\otimes]$ is of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

Convention
[]: Syntax →Semantics

$$\begin{array}{rcl} \mathcal{B}[\mathbf{true}\ ]\sigma & = & \text{tt} \\ \mathcal{B}[a_1\ rop\ a_2]\sigma & = & [rop](\mathcal{A}[a_1]\sigma, \mathcal{A}[a_2]\sigma) \\ & & \text{where } [rop] : \mathbb{Z} \times \mathbb{Z} \to T \\ \mathcal{B}[b_1\ bop\ b_2]\sigma & = & [bop](\mathcal{B}[b_1]\sigma, \mathcal{B}[b_2]\sigma) \\ & & \text{where } [bop] : T \times T \to T \\ & & \text{and } [\wedge] = \begin{array}{c|c|c} & \text{tt} & \text{ff} \\ \hline \text{tt} & \dots & \dots \\ \text{ff} & \dots & \dots \end{array} \\ \mathcal{B}[\neg b]\sigma & = & \begin{cases} \text{tt} & \text{if } \mathcal{B}[b]\sigma = \text{ff} \\ \text{ff} & \text{if } \mathcal{B}[b]\sigma = \text{tt} \end{cases} \end{array}$$

$$\begin{array}{rcl} \mathcal{A}[x+5]\{x \overset{\sigma}{\mapsto} z\} & = & [+](\mathcal{A}[x]\sigma, \mathcal{A}[5]\sigma) \\ & = & [+](7,5) = 12 \end{array}$$

Fact: $\mathcal{A}$ and $\mathcal{B}$ are unique total functions because

1. each syntactic case is covered
2. the arguments of $\mathcal{A}$ / $\mathcal{B}$ on the right hand side are strictly smaller than on the left hand side (i.e. computation terminates)
3. for each expression, only one clause for $\mathcal{A}$ / $\mathcal{B}$ applies

Definition of $\mathcal{A}$ / $\mathcal{B}$ is by *primitive recursion*.
Corresponding proof principle: *structural induction*.

[2] [WINSKEL93]: $\sigma[n/x]$

2 Operational Semantics

Aim: abstract mathematical model of computation steps
Method: inference rules (derivation rules)

$$\frac{\bullet \quad \bullet}{\bullet} \rightsquigarrow \text{Prolog}$$

2.1 Big-step semantics

(Natural semantics, Evaluation semantics)

Idea: relation (transition relation)
$\langle statement, state\rangle \to state$
$\langle s, \sigma\rangle \to \sigma'$

Formally: $\to \ \subseteq Stm \times \Sigma \times \Sigma$
Notation: $(s, \sigma, \sigma') \in \to$ is an abbreviation to $\langle s, \sigma\rangle \to \sigma'$

Rules for $\to$

$$\left.\begin{array}{c} \overline{\langle \mathbf{skip}, \sigma\rangle \to \sigma} \\ \\ \langle x := a, \sigma\rangle \to \sigma[x \mapsto \mathcal{A}[a]\sigma] \end{array}\right\} \text{axioms}$$

$$\frac{\langle s_1, \sigma_1\rangle \to \sigma_2 \qquad \langle s_2, \sigma_2\rangle \to \sigma_3}{\langle s_1; s_2, \sigma\rangle \to \sigma_3}$$

$$\frac{\mathcal{B}[b]\sigma = \mathrm{tt} \qquad \langle s_1, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma\rangle \to \sigma'} \text{ analogously for ff}, s_2$$

w = **while** b **do** s

$$\frac{\mathcal{B}[b]\sigma = \mathrm{ff}}{\langle w, \sigma\rangle \to \sigma} \qquad \frac{\mathcal{B}[b]\sigma = \mathrm{tt} \qquad \langle s, \sigma\rangle \to \sigma' \qquad \langle w, \sigma'\rangle \to \sigma''}{\langle w, \sigma\rangle \to \sigma''}$$

Remarks:

1. General format of rules

$$\frac{premise_1 \quad \dots \quad premise_n}{conclusion} \text{ (name)}$$

2. Axiom = rule without premises
3. Rules can be composed to form derivation trees.
4. Directly executable in Prolog.

$$\frac{\overline{\langle x := 1, \sigma_0\rangle \to \sigma_1} \qquad \overline{\langle y := x, \sigma_1\rangle \to \sigma_2}}{\langle x := 1;\ y := x, \sigma_0\rangle \to \sigma_2}$$

$$\begin{array}{lcl} \sigma_0 & = & \{x \mapsto 0, y \mapsto 0\} \\ \sigma_1 & = & \sigma_0[x \mapsto 1] = \{x \mapsto 1, y \mapsto 0\} \\ \sigma_2 & = & \sigma_1[y \mapsto x] = \{x \mapsto 1, y \mapsto 1\} \end{array}$$

$w = \textbf{while } x \neq 1 \textbf{ do } x := x+1$
$\sigma_n = \{x \mapsto n\}$

$$\dfrac{\mathcal{B}[x\neq 1]\sigma_0 = \text{tt} \qquad \langle x := x+1, \sigma_0\rangle \to \sigma_1 \qquad \dfrac{\mathcal{B}[x\neq 1]\sigma_1 = \text{ff}}{\langle w,\sigma_1\rangle \to \sigma_1}}{\langle w,\sigma_0\rangle \to \sigma_1}$$

$\langle w,\sigma_2\rangle \to$ has no final state
$\nexists \sigma'.\langle w,\sigma_2\rangle \to \sigma'$

Fact: In a big-step semantics, nontermination of s starting in σ is expressed by the non-existence of some σ' such that $\langle s,\sigma\rangle \to \sigma'$

Program equivalence with respect to $\to$

10/24/2001

$$s_1 \sim s_2 \iff \forall \sigma,\sigma'.\langle s_1,\sigma\rangle \to \sigma' \Leftrightarrow \langle s_2,\sigma\rangle \to \sigma'$$

Lemma:

$$w \sim \textbf{if } b \textbf{ then } s;\ w \textbf{ else skip } =: w'$$

Proof: Show $\langle w,\sigma\rangle \to \sigma' \Leftrightarrow \langle w',\sigma\rangle \to \sigma'$ by case distinction on $\mathcal{B}[b]\sigma$ and analysis of the last inference rule used.

1. $\mathcal{B}[b]\sigma = \text{ff}$

$$\dfrac{}{\langle w,\sigma\rangle \to \sigma} \Leftrightarrow \dfrac{\langle \textbf{skip},\sigma\rangle \to \sigma}{\langle w',\sigma\rangle \to \sigma}$$

Motto: Transformation of derivations

2. $\mathcal{B}[b]\sigma = \text{tt}$

$$\dfrac{\dfrac{D_1}{\langle s,\sigma\rangle \to \sigma''} \qquad \dfrac{D_2}{\langle w,\sigma''\rangle \to \sigma'}}{\langle w,\sigma\rangle \to \sigma'} \Leftrightarrow \dfrac{\dfrac{\dfrac{D_1}{\langle s,\sigma\rangle \to \sigma''} \qquad \dfrac{D_2}{\langle w,\sigma''\rangle \to \sigma'}}{\langle s;\ w,\sigma\rangle \to \sigma'}}{\langle w',\sigma\rangle \to \sigma'}$$

Theorem: WHILE is deterministic

$$\langle s,\sigma\rangle \to \sigma_1 \wedge \langle s,\sigma\rangle \to \sigma_2 \Rightarrow \sigma_1 = \sigma_2$$

Proof: by induction on s
1 case: $s = \textbf{while } b \textbf{ do } s_0$ and $\mathcal{B}[b]\sigma = \text{tt}$

$$\dfrac{\langle s_0,\sigma\rangle \to \sigma_1' \qquad \langle s,\sigma_1'\rangle \to \sigma_1}{\langle s,\sigma\rangle \to \sigma_1} \wedge \dfrac{\langle s_0,\sigma\rangle \to \sigma_2' \qquad \langle s,\sigma_2'\rangle \to \sigma_2}{\langle s,\sigma\rangle \to \sigma_2} \overset{I.H.}{\Rightarrow} \sigma_1' = \sigma_2' \overset{I.H.}{\Rightarrow} \sigma_1 = \sigma_2 \ \lightning$$

(no structural induction, as s is not smaller than s); but: rule induction over number of derivation steps.

Problem: while - rule is not compositional with respect to the syntax; **while** uses **while** .
$\left[\text{Compositional: } \mathcal{A}[a_1+a_2]\sigma = [''+''](\mathcal{A}[a_1]\sigma, \mathcal{A}[a_2]\sigma)\right]$

2.2 Rule induction - an introduction

A set of rules $\dfrac{a_1 \in X \quad \ldots \quad a_n \in X}{a \in X}$ defines X *inductively* as the least (smallest) set that satisfies all rules. (least = only those elements that are provably in the set = no junk)

Example: $0 \in N \quad \dfrac{n \in N}{n+1 \in N}$ defines $\mathbb{N}$

but $\mathbb{R}$ also satisfies these rules! $\mathbb{N}$ is the least set. Associated proof principle: rule induction.

Idea: $P(x)$ holds[3] for all $x \in N$ if P is *"preserved by all rules"*: $P(0)$ and $P(n) \Rightarrow P(n+1)$

In general: $P(a_1) \wedge \ldots \wedge P(a_n) \Rightarrow P(a)$

Proof principle:
$[x \in X \Rightarrow P(x)]$ if P is preserved by all rules $(\forall x \in X . P(x))$
Motivation: induction on the size of the derivation.

Some easy examples

$$0 \in N \qquad (R0)$$

$$\frac{n \in N}{n+1 \in N} \qquad (R1)$$

Lemma: $n \in N \Rightarrow \underbrace{n+n \in N}_{P(n)}$

Proof: P is preserved by $R0$ and $R1$:

$P(0) = (0+0 \in N)\checkmark$ (by R0)

$$\begin{array}{ccc} P(n) & \overset{?}{\Rightarrow} & P(n+1) \\ \shortparallel & & \shortparallel \\ n+n \in N & & (n+1)+(n+1) \in N \\ \searrow & & \nearrow \\ & (n+n)+1 \in N & \end{array}$$

Lemma: $m \in N \wedge n \in N \Rightarrow m+n \in N$

$$m \in N \Rightarrow (\forall n. \underbrace{n \in N \Rightarrow m+n \in N}_{P(m)})$$

Proof: by rule induction

$$\begin{array}{lll} R0 & : & P(0) = (n \in N \Rightarrow 0+n \in N)\checkmark \\ R1 & : & P(m) \overset{?}{=} P(m+1) \\ & & \text{Given: } n \in N \Rightarrow m+n \in N \quad \text{(I.H.)} \rightarrow P(m) \\ & & \text{Show: } n \in N \Rightarrow (m+1)+n \in N \end{array}$$

[3] "is provable"

Transitive reflexive closure

Given a relation $\to$, define $\xrightarrow{*}$:

$$\frac{}{x \xrightarrow{*} x} \quad R0$$

$$\frac{x \xrightarrow{*} y \qquad \overbrace{y \to z}^{\text{side condition}}}{x \xrightarrow{*} z} \quad R1$$

Rule induction for $\xrightarrow{*}$

$x \xrightarrow{*} y \Rightarrow P(x,y)$ if

1. $(\forall x)P(x,x)$
2. $P(x,y) \wedge y \to z \Rightarrow P(x,z)$

Lemma: $x \to y \Rightarrow x \xrightarrow{*} y$ (derived rule)[4]

Proof: $$\text{R1}\ \frac{\text{R0}\ \frac{}{x \xrightarrow{*} x} \qquad x \to y}{x \xrightarrow{*} y}$$

Theorem: $x \to y \wedge y \xrightarrow{*} z \Rightarrow x \xrightarrow{*} z \equiv y \xrightarrow{*} z \Rightarrow \underbrace{(x \to y \Rightarrow x \xrightarrow{*} z)}_{P(y,z)}$

Proof: by rule induction

Rename $b \xrightarrow{*} c \Rightarrow \underbrace{(a \to b \Rightarrow a \xrightarrow{*} c)}_{P(b,c)}$

$R0$: $P(x,x) = (a \to x \Rightarrow a \xrightarrow{*} x)$ Lemma ✓

$R1$: $P(x,y) \wedge y \to z \overset{?}{\Rightarrow} P(x,z)$

Given: $a \to x \Rightarrow a \xrightarrow{*} y$
Given: $y \to z$
$\Big\} \overset{R1}{\Rightarrow} a \xrightarrow{*} z$

Show: $a \to x \Rightarrow a \xrightarrow{*} z$

$$\text{R}\ \frac{a_1 \in X \quad \ldots \quad a_n \in X \quad \overbrace{C_1 \ldots C_m}^{\text{side conditions; dont't mention } X}}{a \in X}$$

P is preserved by R if

$$P(a_1) \wedge \ldots \wedge P(a_n) \wedge C_1 \wedge \ldots \wedge C_m \Rightarrow P(a)$$

$\langle _,_ \rangle \to _ \subseteq Stm \times \Sigma \times \Sigma$
$\langle s,\sigma \rangle \to \sigma' = (s,\sigma,\sigma') \in \langle _,_ \rangle \to _$

10/29/2001

[4] no induction, only composition of rules

To show: $\forall s, \sigma, \sigma' : \langle s, \sigma \rangle \to \sigma' \Rightarrow P(s, \sigma, \sigma')$

We have to prove

- $\forall \sigma . P(\textbf{skip}\ , \sigma, \sigma)$
- $\forall x, a, \sigma . P\left(x := a, \sigma, \sigma\left[x \mapsto [a]_\sigma\right]\right)$
- $\forall s_1, s_2, \sigma, \sigma', \sigma'' . P(s_1, \sigma, \sigma'') \wedge P(s_2, \sigma'', \sigma') \Rightarrow P(s_1;\ s_2, \sigma, \sigma')$
- $\forall b, s_1, s_2, \sigma, \sigma' . P(s_1, \sigma, \sigma') \wedge [b]_\sigma = \text{tt} \Rightarrow P(\textbf{if}\ b\ \textbf{then}\ s_1 \textbf{else}\ s_2, \sigma, \sigma')$
  $\forall b, s_1, s_2, \sigma, \sigma' . P(s_2, \sigma, \sigma') \wedge [b]_\sigma = \text{ff} \Rightarrow P(\textbf{if}\ b\ \textbf{then}\ s_1 \textbf{else}\ s_2, \sigma, \sigma')$
- $\forall b, s, \sigma . [b]_\sigma = \text{ff} \Rightarrow P(\textbf{while}\ b\ \textbf{do}\ s, \sigma, \sigma)$
  $\forall b, s, \sigma, \sigma', \sigma'' . [b]_\sigma = \text{tt} \wedge P(w, \sigma'', \sigma') \wedge P(s, \sigma, \sigma'') \Rightarrow P(w, \sigma, \sigma')$

WHILE is deterministic:

$$\forall s, \sigma, \sigma', \sigma'' . \langle s, \sigma \rangle \to \sigma' \wedge \langle s, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma'' \tag{1}$$

Proof: by rule induction on $\to$

- transforming (1):

$$\forall s, \sigma, \sigma' . \langle s, \sigma \rangle \to \sigma' \Rightarrow \underbrace{(\forall \sigma'' . \langle s, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma'')}_{P(s,\sigma,\sigma')}$$

- show $P(\textbf{skip}\ , \sigma, \sigma) = \forall \sigma'' . \langle \textbf{skip}\ , \sigma \rangle \to \sigma'' \Rightarrow \sigma = \sigma''$ (by **skip** - rule)
- show ...
- show $[b]_\sigma = \text{tt} \wedge P(s, \sigma, \sigma_0) \wedge P(w, \sigma_0, \sigma') \Rightarrow P(w, \sigma, \sigma')$

  assume $\begin{array}{l} \forall \sigma'' . \langle s, \sigma \rangle \to \sigma'' \Rightarrow \sigma_0 = \sigma'' \quad (IH_s) \\ \forall \sigma'' . \langle w, \sigma_0 \rangle \to \sigma'' \Rightarrow \sigma' = \sigma'' \quad (IH_w) \end{array}$

  show $\forall \sigma'' . \langle w, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma''$ (σ'' fixed, but arbitrary.)
  assume $\langle w, \sigma \rangle \to \sigma''$
  show $\sigma' = \sigma''$

  Derivation of $\langle w, \sigma \rangle \to \sigma''$

$$\frac{[b]_\sigma = \text{tt} \qquad \overbrace{\langle s, \sigma \rangle \to \sigma_1}^{\mathcal{S}} \qquad \langle w, \sigma_1 \rangle \to \sigma''}{\langle w, \sigma \rangle \to \sigma''}$$

  from $\mathcal{S}$ and IH_s we have $\sigma_0 = \sigma_1 \Rightarrow \langle s, \sigma \rangle \to \sigma_0$ and $\langle w, \sigma_0 \rangle \to \sigma''$
  with IH_w we have $\sigma' = \sigma''$. ✓

2.3 Structural operational semantics

(transition semantics, small-step semantics)

Idea: not one big step, but many small steps.

Format: $\langle s,\sigma\rangle \rightarrow_1 \gamma$ where

- either γ is of the form $\langle s',\sigma'\rangle$;
  then the execution has not terminated and s' is the "remaining" statement to execute
- or γ is of the form σ';
  then the execution has terminated in state σ'

Definition: $\langle s,\sigma\rangle$ is *stuck* if $\nexists\gamma.\langle s,\sigma\rangle \rightarrow_1 \gamma$

Rules for $\rightarrow_1$:

- $\langle \textbf{skip}\ ,\sigma\rangle \rightarrow_1 \sigma$
- $\langle x := a,\sigma\rangle \rightarrow_1 \sigma[x \mapsto \mathcal{A}[a]\sigma]$
- $\dfrac{\langle s_1,\sigma\rangle \rightarrow_1 \langle s_1',\sigma'\rangle}{\langle s_1;s_2,\sigma\rangle \rightarrow_1 \langle s_1';s_2,\sigma'\rangle}$
- $\dfrac{\langle s_1,\sigma\rangle \rightarrow_1 \sigma'}{\langle s_1;s_2,\sigma\rangle \rightarrow_1 \langle s_2,\sigma'\rangle}$
- $\dfrac{\mathcal{A}[b]\sigma = \text{tt} \ /\ \text{ff}}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\sigma\rangle \rightarrow_1 \langle s_1,\sigma\rangle \ /\ \langle s_2,\sigma\rangle}$
- $\dfrac{}{\langle w,\sigma\rangle \rightarrow_1 \langle \textbf{if}\ b\ \textbf{then}\ s;w\ \textbf{else}\ \textbf{skip}\ ,\sigma\rangle}$

Definition:
A *derivation sequence* for $\langle s,\sigma\rangle = \gamma_0$ is either a finite sequence $\gamma_0 \rightarrow_1 \gamma_1 \rightarrow_1 \ldots \rightarrow_1 \gamma_k$ such that $\gamma_k \in \Sigma$ or γ_k is stuck (*termination*) or an infinite sequence $\gamma_0 \rightarrow_1 \gamma_1 \rightarrow_1 \ldots$ (*nontermination*).
Each step $\gamma_i \rightarrow_1 \gamma_{i+1}$ must be justified by a derivation tree. Usual notation $\xrightarrow{n}_1$ / $\xrightarrow{*}_1$ means n / finitely many steps of execution.

Example: $\sigma = \{x \mapsto 1, y \mapsto 2\}$

$$\begin{aligned}\langle (z := x; x := y); y := z,\sigma\rangle &\rightarrow_1 \langle x := y; y := z, \sigma[z \mapsto 1]\rangle \\ &\rightarrow_1 \langle y := z, \sigma[z \mapsto 1, x \mapsto 2]\rangle \\ &\rightarrow_1 \sigma[z \mapsto 1, x \mapsto 2, y \mapsto 1]\end{aligned} \tag{2}$$

Justify each step with derivation tree:

$$(2): \dfrac{\dfrac{\vdots \qquad \vdots}{\langle z := x; x := y,\sigma\rangle \rightarrow_1 \langle x := y,\sigma_z\rangle}}{\langle (z := x; x := y); y := z,\sigma\rangle \rightarrow_1 \langle x := y; y := z,\sigma_0\rangle}$$

11/05/2001

Example: $w = \textbf{while}\ x \neq 1\ \textbf{do}\ x := x+1 \quad \sigma_n = \{x \mapsto n\}$

$$\langle w,\sigma_0\rangle \quad \rightarrow_1 \quad \langle \underbrace{\textbf{if}\ x \neq 1\ \textbf{then}\ x := x+1; w\ \textbf{else}\ \textbf{skip}}_{\textbf{if}}\ ,\sigma_0\rangle$$

$$\begin{array}{rcl} \langle \textbf{if}, \sigma_0 \rangle & \to_1 & \langle x := x+1; w, \sigma_0 \rangle \\ \langle x := x+1; w, \sigma_0 \rangle & \to_1 & \langle w, \sigma_1 \rangle \\ \langle w, \sigma_1 \rangle & \to_1 & \langle \textbf{if}, \sigma_1 \rangle \\ \langle \textbf{if}, \sigma_1 \rangle & \to_1 & \langle \textbf{skip}, \sigma_1 \rangle \\ \langle \textbf{skip}, \sigma_1 \rangle & \to_1 & \sigma_1 \end{array}$$

$$\begin{array}{l} \langle w, \sigma_2 \rangle \to_1 \langle \textbf{if}, \sigma_2 \rangle \to_1 \langle x := x+1; w, \sigma_2 \rangle \to_1 \langle w, \sigma_3 \rangle \\ \langle w, \sigma_3 \rangle \to_1 \dots \end{array}$$

Lemma: There are no stuck $\langle s, \sigma \rangle$.

Proof: by induction on s (structural induction)

- $s = \textbf{skip}$ $\to$ not stuck
- $s = s_1; s_2$ IH: $\exists \gamma. \langle s_1, \sigma \rangle \to_1 \gamma$
  Case distinction on γ
  - $\gamma = \sigma' \Rightarrow \langle s_1; s_2, \sigma \rangle \to_1 \langle s_2, \sigma' \rangle$ ✓
  - $\gamma = \langle s_1', \sigma' \rangle \Rightarrow \langle s_1; s_2, \sigma \rangle \to_1 \langle s_1'; s_2, \sigma \rangle$ ✓

Predominant proof principle for $\xrightarrow{*}_1$: *induction* on the length of the derivation sequence.

Lemma: If $\langle s_1; s_2, \sigma \rangle \xrightarrow{n}_1 \sigma''$ then there exist σ', i, j such that

$$\langle s_1, \sigma \rangle \xrightarrow{i}_1 \sigma' \text{ and } \langle s_2, \sigma' \rangle \xrightarrow{j}_1 \sigma'' \text{ and } n = i + j$$

Proof: by induction on n
$n = 0:$ ✓
$n \to n+1$: assume the lemma holds for n, then prove for $n+1$:

$$\langle s_1; s_2, \sigma \rangle \to_1 \gamma \xrightarrow{n}_1 \sigma''$$

$\to$ Case distinction on first step

1. $$\frac{\langle s_1, \sigma \rangle \to_1 \sigma_1}{\langle s_1; s_2, \sigma \rangle \to_1 \underbrace{s_2, \sigma_1}_{\gamma}}$$
   Then the lemma holds for $\sigma' = \sigma_1, i = 1, j = n$ ✓
2. $$\frac{\langle s_1, \sigma \rangle \to_1 \langle s_1', \sigma_1 \rangle}{\langle s_1; s_2, \sigma \rangle \to_1 \langle \underbrace{s_1', s_2}_{\gamma}, \sigma_1 \rangle}$$
   By IH:
   $\langle s_1', \sigma_1 \rangle \xrightarrow{i_x}_1 \sigma_x$

$\langle s_2, \sigma_x \rangle \overset{j_x}{\rightarrow}_1 \sigma'' \quad i_x + j_x = n$

Then the lemma holds for $\sigma' = \sigma_x, i = i_x + 1, j = j_x$
$\langle s_1, \sigma \rangle \overset{i}{\rightarrow}_1 \sigma'$ because $\langle s_1, \sigma \rangle \rightarrow_1 \langle s_1', \sigma_1 \rangle \overset{i_x}{\rightarrow}_1 \sigma'$ ✓

Theorem: (Agreement of big-step and small-step semantics)

$$\langle s, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle s, \sigma \rangle \overset{*}{\rightarrow}_1 \sigma'$$

This implies

- Termination in one semantics iff. termination in the other semantics, and with same result
- Looping in one semantics iff. looping in the other semantics.

Proof: "⇒": exercise (Übungsblatt 3)
"⇐": $\langle s, \sigma \rangle \overset{n}{\rightarrow}_1 \sigma' \Rightarrow \langle s, \sigma \rangle \rightarrow \sigma'$

Proof by induction on n:
IH: Assume that it works for all $m < n$, then show it holds for n

$n > 0$, because $\langle s, \sigma \rangle \overset{0}{\rightarrow}_1 \sigma'$ is not possible. Case distinction on s:

- $s = \textbf{skip}$: $\sigma = \sigma', \langle \textbf{skip}, \sigma \rangle \rightarrow \sigma$ ✓
- $s = (x := a) : \sigma' = \sigma[x \mapsto \mathcal{A}[a]\sigma], \langle x := a, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[a]\sigma]$
- $s = s_1; s_2$: show $\langle s_1; s_2, \sigma \rangle \rightarrow \sigma'$

$$\begin{array}{lll} \text{lemma} & \Rightarrow & \exists \sigma'', i, j. \langle s_1, \sigma \rangle \overset{i}{\rightarrow}_1 \sigma'' \wedge \langle s_2, \sigma'' \rangle \overset{j}{\rightarrow}_1 \sigma' \wedge n = i + j \\ & \overset{IH}{\Rightarrow} & \langle s_1, \sigma \rangle \rightarrow \sigma'' \wedge \langle s_2, \sigma'' \rangle \rightarrow \sigma' \\ & \overset{;-rule}{\Rightarrow} & \langle s_1; s_2, \sigma \rangle \rightarrow \sigma' \quad ✓ \end{array}$$

- $s = \textbf{if}$ ✓ (similar as ;)
- $w = \textbf{while}\ b\ \textbf{do}\ s_0 = s$

$$\langle w, \sigma \rangle \rightarrow_1 \langle \underbrace{\textbf{if}\ b\ \textbf{then}\ s_0; w\ \textbf{else skip}}_{w'}, \sigma \rangle \overset{n-1}{\rightarrow}_1 \sigma'$$

$$\begin{array}{ll} \overset{IH}{\Rightarrow} & \langle w', \sigma \rangle \rightarrow \sigma' \\ \Rightarrow & \langle s, \sigma \rangle \rightarrow \sigma' \text{ because } w \sim w' \end{array}$$

2.4 Extensions of the WHILE language

- Nondeterminism
- Parallelism
- Blocks
- Procedures

} comparison of big/small step semantics

2.4.1 Nondeterminism

New statement: s_1 **or** s_2

Example:

$$\underbrace{\langle \overbrace{x := 1}^{s_1} \textbf{ or } \overbrace{(x := 2; x := x+4)}^{s_{24}}}_{s_{124}}, \sigma\rangle \rightarrow \begin{cases} x = 1 \\ x = 6 \end{cases}$$

Big-step rules

$$\frac{\langle s_1, \sigma\rangle \rightarrow \sigma'}{\langle s_1 \textbf{ or } s_2, \sigma\rangle \rightarrow \sigma'} \qquad \frac{\langle s_2, \sigma\rangle \rightarrow \sigma'}{\langle s_1 \textbf{ or } s_2, \sigma\rangle \rightarrow \sigma'}$$

Example:

$$\frac{\overline{\langle s_1, \sigma\rangle \rightarrow \sigma[x \mapsto 1]}}{\langle s_{124}, \sigma\rangle \rightarrow \sigma[x \mapsto 1]} \qquad \frac{\dfrac{\overline{\langle s_2, \sigma\rangle \rightarrow \sigma[x \mapsto 2]} \quad \overline{\langle s_4, \sigma[x \mapsto 2]\rangle \rightarrow \sigma[x \mapsto 6]}}{\langle s_{24}, \sigma\rangle \rightarrow \sigma[x \mapsto 6]}}{\langle s_{124}, \sigma\rangle \rightarrow \sigma[x \mapsto 6]}$$

$$\langle x := 1 \textbf{ or } (\underbrace{\textbf{while true do skip}}_{\Omega}), \sigma\rangle \rightarrow \ ? \ \ \sigma[x \mapsto 1]$$

Small-step rules:

11/07/2001

$$\begin{array}{lcl} \langle s_1 \textbf{ or } s_2, \sigma\rangle & \rightarrow_1 & \langle s_1, \sigma\rangle \\ \langle s_1 \textbf{ or } s_2, \sigma\rangle & \rightarrow_1 & \langle s_2, \sigma\rangle \end{array}$$

Example:

$$\begin{array}{lcl} \langle s_{124}, \sigma\rangle & \rightarrow_1 & \langle s_{24}, \sigma\rangle \rightarrow_1 \langle s_4, \sigma[x \mapsto 2]\rangle \rightarrow_1 \sigma[x \mapsto 6] \\ \langle s_{124}, \sigma\rangle & \rightarrow_1 & \langle s_1, \sigma\rangle \rightarrow_1 \sigma[x \mapsto 1] \end{array}$$

$$\begin{array}{lcl} \langle x := 1 \textbf{ or } \Omega, \sigma\rangle & \rightarrow_1 & \langle x := 1, \sigma\rangle \rightarrow_1 \sigma[x \mapsto 1] \\ \langle x := 1 \textbf{ or } \Omega, \sigma\rangle & \rightarrow_1 & \langle \Omega, \sigma\rangle \rightarrow_1 \langle \textbf{if true then skip }; \Omega \textbf{ else skip }, \sigma\rangle \\ & \rightarrow_1 & \langle \textbf{skip }; \Omega, \sigma\rangle \rightarrow_1 \langle \Omega, \sigma\rangle \rightarrow_1 \ldots \end{array}$$

Comparison:

- big-step semantics hides nontermination:

$$x := 1 \textbf{ or } \Omega \quad \sim \quad x := 1$$

- small-step semantics does not hide termination:

$$\begin{array}{rl} x := 1 & \text{has no infinite derivation sequence} \\ x := 1 \textbf{ or } \Omega & \text{has an infinite derivation sequence} \end{array}$$

2.4.2 Parallelism

New statement $s_1 \parallel s_2$
Semantics: interleaving execution (s_1 and s_2 "alternate")

Small-step rules

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \langle s_1', \sigma'\rangle}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_1' \parallel s_2, \sigma'\rangle} \qquad \frac{\langle s_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_2, \sigma'\rangle}$$

$$\frac{\langle s_2, \sigma\rangle \rightarrow_1 \langle s_2', \sigma'\rangle}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_1 \parallel s_2', \sigma'\rangle} \qquad \frac{\langle s_2, \sigma\rangle \rightarrow_1 \sigma'}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_1, \sigma'\rangle}$$

Example: $s_{124} := x := 1 \parallel (x := 2; x := x + 4) \qquad \Rightarrow x = 1, 5, 6$

$$\langle s_{124}, \sigma\rangle \quad \rightarrow_1 \quad \langle s_{24}, \sigma[x \mapsto 1]\rangle \rightarrow_1 \ldots \rightarrow_1 \sigma[x \mapsto 6]$$

$$\begin{array}{rcl} \langle s_{124}, \sigma\rangle & \rightarrow_1 & \langle x := 1 \parallel x := x + 4, \sigma[x \mapsto 2]\rangle \\ & \rightarrow_1 & \langle x := 1, \sigma[x \mapsto 6]\rangle \rightarrow_1 \sigma[x \mapsto 1] \end{array}$$

$$\begin{array}{rcl} \langle s_{124}, \sigma\rangle & \rightarrow_1 & \langle x := 1 \parallel x := x + 4, \sigma[x \mapsto 2]\rangle \\ & \rightarrow_1 & \langle x := x + 4, \sigma[x \mapsto 1]\rangle \rightarrow_1 \sigma[x \mapsto 5] \end{array}$$

Big-step rules

$$\frac{?}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow \sigma'}$$

Comparison:

- Big-steps hide intermediate states $\Rightarrow$ parallelism impossible
- Small-steps show intermediate states $\Rightarrow$ parallelism possible

A language extension is called *modular* if

- new rules are added, but
- old rules stay unchanged

or is modular: $\rightarrow$ and $\rightarrow_1$
$\parallel$ is modular: $\rightarrow_1$

Not all language extensions are modular!

2.4.3 Blocks (Local variables)

New statement: $\{\mathbf{var}\ x := a; s\}$ **Idea:** x is local variable in s.

Example:

$$\langle x := 1;\ \{\mathbf{var}\ x := 5; \mathbf{skip}\ \}, \sigma\rangle \rightsquigarrow \sigma[x \mapsto 1]$$

$$\langle y := 1;\ \{\mathbf{var}\ x := 5; \mathbf{skip}\ \}, \sigma\rangle \rightsquigarrow \sigma[y \mapsto 1]$$

Big-step rules

$$\frac{\langle x := a; s, \sigma\rangle \to \sigma'}{\langle\{\mathbf{var}\ x := a; s\}, \sigma\rangle \to \sigma'[x \mapsto \sigma(x)]}$$

Small-step rules

$$\langle\{\mathbf{var}\ x := a; s\}, \sigma\rangle \to_1 \langle x := a; s; x := \underbrace{\sigma(x)}_{\in\mathbb{Z}}, \sigma\rangle$$

2.4.4 Procedures

New statements: $\{\mathbf{var}\ x := a; s\}$
$\{\mathbf{proc}\ (p = s_1); s_2\}$
$\mathbf{call}\ p$
Parameterless procedures!

Example:

```
{ var x:=0;
  { proc p = x:=2*x;
    { proc q = call p;
      { var x:=5;
        { proc p = x:=x+1;
          call q; y:=x
}}}}}
```

Final value of y? $\to 5$!

Static scope: Name x/p refers to the closest declaration of x/p in the *text*.
Dynamic scope: Name x/p refers to the most recent definition of x/p during *execution*.

- static scope for **var** and for **proc** : $y = 5$
- dynamic scope for **var** and static scope for **proc** :

$$x := 0; x := 5; x := 2 * x; y := x$$

$\to y = 10, x = 0$ after 4th "}"; at the end of "**call** $q; y := x$", x is not defined.

- dynamic scope for **var** and **proc** :

$$x : 0; x := 5; \overset{x:=x+1}{(x := 6)}; y := x \to y = 6$$

Dynamic scope for proc and var

11/12/2001

Procedure environment: $Penv = \underbrace{Pname}_{name} \to \underbrace{Stm}_{body}$

$pe \in Penv$
Format of transitions: $pe \vdash \langle s, \sigma \rangle \rightarrow \sigma'$
"In the context of pe, execution of s starting in σ leads to σ'."

Rules for **while** -language: add $pe \vdash$ everywhere.

$$\frac{pe \vdash \langle s_1, \sigma_1 \rangle \rightarrow \sigma_2 \qquad pe \vdash \langle s_2, \sigma_2 \rangle \rightarrow \sigma_3}{pe \vdash \langle s_1; s_2, \sigma_1 \rangle \rightarrow \sigma_3}$$

$$\frac{pe \vdash \langle s, \sigma[x \mapsto \mathcal{A}[a]\sigma] \rangle \rightarrow \sigma'}{pe \vdash \langle \{\mathbf{var}\ x := a;\ s\}, \sigma \rangle \rightarrow \sigma'[x \mapsto \sigma(x)]}$$

$$\frac{pe[p \mapsto s_1] \vdash \langle s_2, \sigma \rangle \rightarrow \sigma'}{pe \vdash \langle \{\mathbf{proc}\ p = s_1; s_2\}, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{pe \vdash \langle pe(p), \sigma \rangle \rightarrow \sigma'}{pe \vdash \langle \mathbf{call}\ p, \sigma \rangle \rightarrow \sigma'}$$

```
s0  { var x:=0;
s1    { proc p = x:=2*x;
s2      { proc q = call p;
s3        { var x:=5;
s4          { proc p = x:=x+1;
s5              call q; y:=x;
    }}}}}
```

Example: $\{\} \vdash \langle s_0, \sigma \rangle \rightarrow ? \qquad \sigma_i = \sigma[x \mapsto i]$

$$\begin{array}{c}
\{\} \vdash \langle s_0, \sigma \rangle \rightarrow \sigma_0[y \mapsto 6] \\ \hline
\{\} \vdash \langle s_1, \sigma_0 \rangle \rightarrow \sigma_0[y \mapsto 6] \\ \hline
\{p \mapsto x := 2 * x\} \vdash \langle s_2, \sigma_0 \rangle \rightarrow \sigma_0[y \mapsto 6] \\ \hline
\{p \mapsto x := 2 * x, q \mapsto \mathbf{call}\ p\} \vdash \langle s_3, \sigma_0 \rangle \rightarrow \sigma_0[y \mapsto 6] \\ \hline
\{p \mapsto x := 2 * x, q \mapsto \mathbf{call}\ p\} \vdash \langle s_4, \sigma_5 \rangle \rightarrow \sigma_6[y \mapsto 6] \\ \hline
\{p \mapsto x := x + 1, q \mapsto \mathbf{call}\ p\} \vdash \langle s_5, \sigma_5 \rangle \rightarrow \sigma_6[y \mapsto 6] \\ \hline
\{p \mapsto x := x + 1, q \mapsto \mathbf{call}\ p\} \vdash \langle \mathbf{call}\ q, \sigma_5 \rangle \rightarrow \sigma_6 \qquad \{\%\} \vdash \langle y := x, \sigma_6 \rangle \rightarrow \sigma_6[x \mapsto 6] \\ \hline
\{p \mapsto x := x + 1, q \mapsto \mathbf{call}\ p\} \vdash \langle \mathbf{call}\ p, \sigma_5 \rangle \rightarrow \sigma_6 \\ \hline
\{p \mapsto x := x + 1, q \mapsto \mathbf{call}\ p\} \vdash \langle x := x + 1, \sigma_5 \rangle \rightarrow \sigma_6
\end{array}$$

Static scope for proc, dynamic scope for var

Idea: remember the static environment (i.e. the program text) for each procedure

$$Penv = Pname \rightarrow \underbrace{(Stm \times Penv)}_{\text{closure}}$$

use $\stackrel{fin}{\rightarrow}$ instead of $\nrightarrow$ to make this work

(s, pe) is closed: procedure names in s are defined in pe ($\leftrightarrow$ open formula: free variables)

The rules: **WHILE** -language and **var** unchanged

$$\frac{pe[p \mapsto (s_1, pe)] \vdash \langle s_2, \sigma \rangle \rightarrow \sigma'}{pe \vdash \langle \{\mathbf{proc}\ p = s_1; s_2\}, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{pe' \vdash \langle s, \sigma \rangle \rightarrow \sigma'}{pe \vdash \langle \mathbf{call}\ p, \sigma \rangle \rightarrow \sigma'} \qquad \text{if } pe(p) = (s, pe')$$

Example: $s = \{\textbf{proc } f = \ldots \textbf{call } f \ldots ; \textbf{call } f\}$

$$\cfrac{\cfrac{\{\} \vdash \langle s, \sigma\rangle \to}{\left\{f \mapsto (\ldots \textbf{call } f \ldots, \{\})\right\} \vdash \langle \textbf{call } f, \sigma\rangle \to}}{\{\} \vdash \langle \ldots \textbf{call } f \ldots\rangle \to \quad \lightning}$$

cannot call f in empty pe!

Corrected call rule

$$\frac{pe'[p \mapsto pe(p)] \vdash \langle s, \sigma\rangle \to \sigma'}{pe \vdash \langle \textbf{call } p, \sigma\rangle \to \sigma'} \text{ if } pe(p) = (s, pe')$$

Mini-example

```
s0  { proc p = x:=0;
s1    { proq q = call p;
s2      { proc p = x:=1;
s3        { call q }}}
```

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\{\} \vdash \langle s_0, \sigma\rangle \to \sigma[x \mapsto 0]}{\underbrace{\{p \mapsto (x := 0, \{\})\}}_{pe_1} \vdash \langle s_1, \sigma\rangle \to \sigma[x \mapsto 0]}}{\underbrace{\{q \mapsto (\textbf{call } p, pe_1), p \to \ldots\}}_{pe_2} \vdash \langle s_2, \sigma\rangle \to \sigma[x \mapsto 0]}}{\{q \mapsto (\textbf{call } p, pe_1), p \mapsto (x := 1, pe_2)\} \vdash \langle \textbf{call } q, \sigma\rangle \to \sigma[x \mapsto 0]}}{pe_1[q \mapsto (\textbf{call } p, pe_1)] \vdash \langle \textbf{call } p, \sigma\rangle \to \sigma[x \mapsto 0]}}{\vdash \langle x := 0, \sigma\rangle \to \sigma[x \mapsto 0] \ \Uparrow}$$

An inductive definition of *Penv*:

- $\{\} \in Penv$
- $\dfrac{pe_1, pe_2 \in Penv \qquad s \in Stm \qquad p \in Pname}{pe_1[p \mapsto (s, pe_2)]}$

An alternative: recursive environments!
Example:

$$p \mapsto (s, \{\ldots, p \mapsto (s, \{\ldots, p \mapsto \ldots\ldots$$

$$\underbrace{pe = \{p \mapsto (s, pe)\}}_{pointer!}$$

$$\frac{pe' \vdash \langle s_2, \sigma\rangle \to \sigma'}{pe \vdash \langle\{\textbf{proc } p = s_1; s_2\}, \sigma\rangle \to \sigma'} \qquad \text{where } pe' = pe[p \mapsto (s_1, pe')] \text{ (recursive definition)}$$

$\Rightarrow$ simpler call rule

$$\frac{pe' \vdash \langle s, \sigma\rangle \to \sigma'}{pe \vdash \langle \textbf{call } p, \sigma\rangle \to \sigma'} \qquad \text{if } pe(p) = (s, pe')$$

Static scope for proc and var

Idea: separate variable names (*Var*) from storage locations (*Loc*)

- *Loc*: some infinite set
- Variable environment: $Venv = Var \stackrel{fin}{\longrightarrow} Loc \quad \ni ve$ [5]
- $Store = Loc \stackrel{fin}{\longrightarrow} \mathbb{Z} \quad \ni \sigma$
- $\sigma \circ ve \in Var \to \mathbb{Z}$
- $Penv = Pname \stackrel{fin}{\longrightarrow} (Stm \times Venv \times Penv)$
- (s, ve, pe): variable x in s refers to location $ve(x)$
- procedure p in s refers to the body $pe(p)$

11/14/2001

Format of transitions: $ve, pe \vdash \langle s, \sigma \rangle \to \sigma'$
Assignment: $ve, pe \vdash \langle x := a, \sigma \rangle \to \sigma[\underbrace{ve(x)}_{\in Loc} \mapsto \mathcal{A}[a](\underbrace{\sigma \circ ve}_{\in \Sigma})]$
Remaining while-rules: add *ve* everywhere

Convention: execution starts with *ve* and σ such that

$$\begin{aligned} range(ve) &= \{l \mid \exists x. ve(x) = l\} \subseteq dom(\sigma) \\ &= \{l \mid \sigma(l) \text{ is defined}\} \end{aligned} \tag{3}$$

Rules will guarantee that (3) is invariant under each transition.
Final value of global variable x: $\sigma'(ve(x)) \quad (x \in dom(ve))$

Rules for **var** :

$$\frac{ve[x \mapsto l], pe \vdash \langle s, \sigma[l \mapsto \mathcal{A}[a](\sigma \circ ve)] \rangle \to \sigma'}{ve, pe \vdash \langle \{\textbf{var } x := a;\ s\}, \sigma \rangle \to \sigma'}$$

where l is some "new" location $\notin dom(\sigma)$
why $\to \sigma'$ and not $\sigma'[l \mapsto undefined]$?
We allow garbage in σ': l is "unreachable".

$$\frac{ve, pe[p \mapsto (s_1, ve, pe)] \vdash \langle s_2, \sigma \rangle \to \sigma'}{ve, pe \vdash \langle \{\textbf{proc } p = s_1;\ s_2\}, \sigma \rangle \to \sigma'}$$

$$\frac{ve', pe'[p \mapsto pe(p)] \vdash \langle s, \sigma \rangle \to \sigma'}{ve, pe \vdash \langle \textbf{call } p, \sigma \rangle \to \sigma'} \qquad \text{if } pe(p) = (s, ve', pe')$$

2.5 A compiler for WHILE

Source language: **WHILE**
Target language: Idealized assembly language (IAL)
Instructions:

- *ASSIGN Var Aexp*
- *JMP* $\mathbb{Z}$ (relative jump)
- *JMPF Bexp* $\mathbb{Z}$ (jump if *Bexp* is false)

Semantics of IAL: Relation on program counter × State

Notation: $\boxed{P \vdash \langle i, \sigma \rangle \to \langle j, \sigma' \rangle}$
where

[5] $(A \stackrel{fin}{\longrightarrow} B$ = all functions $A \rightharpoonup B$ that are defined for only finitely many arguments)

- P: list of instructions
- $i, j \in \mathbb{Z}, \sigma, \sigma' \in \Sigma$

"Execution of instruction i of P in state σ leads to the new program counter j and new state σ'."

Notation: $P_n = (n+1)^{th}$ instruction in P, i.e. P_0 is first instruction. $++$ = concatenation of lists.

Rules:

$$\frac{P_n = ASSIGN\ x\ a}{P \vdash \langle n, \sigma \rangle \to \langle n+1, \sigma[x \mapsto \mathcal{A}[a]\sigma] \rangle}$$

$$\frac{P_n = JMP\ d}{P \vdash \langle n, \sigma \rangle \to \langle n+d, \sigma \rangle}$$

$$\frac{P_n = JMPF\ b\ d \qquad \mathcal{B}[b]\sigma = \text{tt}\ /\ \text{ff}}{P \vdash \langle n, \sigma \rangle \to \langle n+1\ /\ n+d, \sigma \rangle}$$

Notation: $P \vdash c_0 \xrightarrow{*} c_n$ iff. there are $c_1, \ldots, c_{n-1}$ such that

- $P \vdash c_0 \to c_1$ and
- ... and
- $P \vdash c_{n-1} \to c_n$

Execution stops when pc negative or too large because then P_n is undefined.

$$\begin{array}{rcl}
comp : Stm & \to & \text{instruction list} \\
comp\ \textbf{skip} & = & [] \\
comp\ (x := a) & = & [ASSIGN\ x\ a] \\
comp\ (s_1; s_2) & = & comp(s_1) ++ comp(s_2) \\
comp\ (\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2) & = & \textbf{let}\ \begin{array}{l} is_1 = comp\ s_1 \\ is_2 = comp\ s_2 \end{array} \\
 & & \textbf{in}\ [JMPF\ b\ (|is_1| + 2)] ++ is_1 ++ [JMP |is_2| + 1] ++ is_2 \\
comp\ (\textbf{while}\ b\ \textbf{do}\ s) & = & \textbf{let}\ is = comp\ s \\
 & & \textbf{in}\ [JMPF\ b\ |is| + 2] ++ is ++ [JMP - (|is| + 1)]
\end{array}$$

Theorem:

$$\langle s, \sigma \rangle \to \sigma' \Rightarrow comp\ s \vdash \langle 0, \sigma \rangle \xrightarrow{*} \langle |comp\ s|, \sigma' \rangle$$

Proof: by induction on $\langle s, \sigma \rangle \to \sigma'$ (rule induction)

case $s = s_1; s_2$: $\langle s_1, \sigma \rangle \to \overline{\sigma}, \langle s_2, \overline{\sigma} \rangle \to \sigma'$

$$\frac{\langle s_1, \sigma \rangle \to \overline{\sigma} \qquad \langle s_2, \overline{\sigma} \rangle \to \sigma' \qquad \leftarrow \text{assume this}}{\langle s, \sigma \rangle \to \sigma' \Rightarrow Q(s, \sigma, \sigma') \qquad \leftarrow \text{show this}}$$

Abbreviation: $P_i = comp\ s_i$

IH_1: $P_1 \vdash \langle 0, \sigma \rangle \xrightarrow{*} \langle |P_1|, \overline{\sigma} \rangle$

IH_2: $P_2 \vdash \langle 0, \overline{\sigma} \rangle \xrightarrow{*} \langle |P_2|, \sigma' \rangle$

To show: $P_1 ++ P_2 \vdash \langle 0,\sigma\rangle \xrightarrow{*} \langle |P_1|+|P_2|,\sigma'\rangle$

Lemma:
$P \vdash \langle i,\sigma\rangle \xrightarrow{n} \langle j,\sigma'\rangle \Rightarrow P ++ P' \vdash \langle i,\sigma\rangle \xrightarrow{n} \langle j,\sigma'\rangle$
$P \vdash \langle i,\sigma\rangle \xrightarrow{n} \langle j,\sigma'\rangle \Rightarrow P' ++ P \vdash \langle |P'|+i,\sigma\rangle \xrightarrow{n} \langle |P'|+j,\sigma\rangle$

Proof: by induction on n followed by a case distinction on the first rule used.

$$\left.\begin{array}{l} IH_1 + \text{Lemma: } P_1 ++ P_2 \vdash \langle 0,\sigma\rangle \xrightarrow{*} \langle |P_1|,\overline{\sigma}\rangle \\ IH_2 + \text{Lemma: } P_1 ++ P_2 \vdash \langle |P_1|,\overline{\sigma}\rangle \xrightarrow{*} \langle |P_1|+|P_2|,\sigma'\rangle \end{array}\right\} \Rightarrow P_1 ++ P_2 \vdash \langle 0,\sigma\rangle \xrightarrow{*} \langle |P_1|+|P_2|,\sigma'\rangle \text{ qed.}$$

11/19/2001

Theorem: $\langle s,\sigma\rangle \to \sigma' \Rightarrow comp\ s \vdash \langle 0,\sigma\rangle \xrightarrow{*} \langle |comp\ s|,\sigma'\rangle$

Theorem: Let $c = comp\ s$. $c \vdash \langle 0,\sigma\rangle \xrightarrow{*} \langle |c|,\sigma'\rangle \Rightarrow \langle s,\sigma\rangle \to \sigma'$

Proof: by induction on s (because $comp$ is defined recursively)
Case $s = \textbf{while}\ b\ \textbf{do}\ s_0 \Rightarrow$

$$comp\ s \quad = \quad * \ [JMPF\ b\ *] ++ c_0 ++ [JMP\ *]\ *$$

$$c_0 = comp\ s_0$$

We prove $c \vdash \langle 0,\sigma\rangle \xrightarrow{n} \langle |c|,\sigma'\rangle \Rightarrow \langle s,\sigma\rangle \to \sigma'$ by induction on n. We assume it holds for all $m < n$ and prove it for n. Case distinction:

1. $\mathcal{B}[b]\sigma = \text{ff} \Rightarrow \sigma' = \sigma \Rightarrow \langle s,\sigma\rangle \longrightarrow\!\!\!\!\longrightarrow \sigma'$

2. $\mathcal{B}[b]\sigma = \text{tt} \Rightarrow c \vdash \langle 0,\sigma\rangle \to \langle 1,\sigma\rangle \xrightarrow{n-1} \langle |c|,\sigma'\rangle$

   $\exists\overline{\sigma}, n_1, n_2 . c_0 \vdash \langle 0,\sigma\rangle \xrightarrow{n_1} \langle |c_0|,\overline{\sigma}\rangle \wedge$
   $c \vdash \langle |c_0|+1,\overline{\sigma}\rangle \xrightarrow{n_2} \langle |c|,\sigma'\rangle \wedge n_1 + n_2 = n-1$

   (n_1: number of executions of s_0; n_2: number of executions for next iterations[6])

Definition: A list of instructions is *closed* iff. all jumps in c have a "destination" $\in \{0,\ldots,|c|\}$. ("No jump leaves c")

Lemma: (1) Let c be closed.

$$\begin{array}{ll} & p ++ c ++ p' \vdash \langle |p|+i,\sigma\rangle \xrightarrow{n} \langle j,\sigma'\rangle \\ & \text{where } 0 \le i \le |c| \wedge \neg(|p| \le j \le |p|+|c|) \\ \Rightarrow & \exists\overline{\sigma}, n_1, n_2 . c \vdash \langle i,\sigma\rangle \xrightarrow{n_1} \langle |c|,\sigma'\rangle \wedge \\ & p ++ c ++ p' \vdash \langle |p|+|c|,\overline{\sigma}\rangle \xrightarrow{n_2} \langle j,\sigma'\rangle \wedge \\ & n_1 + n_2 = n \end{array}$$

[6] by Lemma(1) and Lemma(2)

Proof: by induction on n.

Lemma: (2) *comp s* is closed.

Proof: by induction on s.

$$
\begin{array}{lcl}
\text{Lemma(1) and Lemma(2)} & \overset{IH_s}{\Rightarrow} & \langle s_0, \sigma\rangle \rightarrow \overline{\sigma} \wedge c \vdash \langle 0, \overline{\sigma}\rangle \overset{n_2-1}{\rightarrow} \langle |c|, \sigma'\rangle \\
 & \overset{IH_1}{\Rightarrow} & \langle s, \overline{\sigma}\rangle \rightarrow \sigma' \\
 & \overset{\text{by while-rule}}{\Rightarrow} & \langle s, \sigma\rangle \rightarrow \sigma' \qquad qed.
\end{array}
$$

3 Types

11/19/2001

Classification

- dynamic type system (out)
  - runtime test if types are ok
  - no help for program development
  - example: LISP
- static type system (in!)
  - analysis of types at compile time
  - only limited precision (e.g. int/bool, but not $\{0,\ldots,42\}$)
  - extremely helpful for program development
  - special case of program analysis and verification

Aim: static, automatic, efficient, precise
guarantee absence of as many runtime errors as possible

3.1 Types for WHILE

Types: **bool** and **int**
$\rightarrow$ Now allowed: $B := I < 0$
$\rightarrow$ Not allowed: $I := J + 1; I := I > J$

Previously: *Aexp* and *Bexp*
Now: $Exp := Aexp \cup Bexp$; type system seperates.

Syntax of *Exp*:

$$e ::= n \mid \textbf{true} \mid \textbf{false} \mid X \mid e\ aop\ e \mid e\ bop\ e \mid e\ rop\ e \mid \neg e$$

"Raw expressions" (e.g. $10 \wedge \neg 5$)

Exp = set of all (raw/untyped) expressions

Modified stm syntax:

- $x := e$
- **if** e **then** ...
- **while** e **do** ...

$\Rightarrow$ New language $\textbf{WHILE}_\textbf{T}$ ("Typed **WHILE** ")

Assumption: "Every variable has a fixed type."

Possible interpretations:

1. Type is fixed for all programs
   (e.g. $I - N$: always integer)
2. Type must be declared in each program
   (e.g. **var** I : integer / **int** I / ...)
3. Type is inferred / computed by the "system"
   $\Rightarrow$ Type interference / Type reconstruction

1 und 2 are *type checking* methods.

3.2 A monomorphic type system

Types $\tau ::= int \mid bool$

Program syntax: $p ::= \underbrace{x_1 : \tau_1; \ldots ; x_n : \tau_n}_{variable\ declarations\ \Gamma} ; s$

Assumption: Γ never declares a variable twice.
*Mono*morphic: a variable has at most one type.

3.2.1 A typesystem for $\mathbf{WHILE_T}$

Types of expression

Want to say "In context Γ, expression e has type τ."

Notation: $\Gamma \vdash e : \tau$

Rules: $\Gamma \vdash n : int, \Gamma \vdash true\ /\ false :\ bool$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ aop\ e2\ \ : \mathbf{int}} \qquad \text{analogously: bop}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ rop\ e2\ \ : \mathbf{bool}} \qquad + \neg e$$

Example: $\Gamma = X : \mathbf{int}\ , Y : \mathbf{bool}$
$\Gamma \vdash (X = X) \wedge \neg Y : \mathbf{bool}$
not $\Gamma \vdash X = Y : \tau$ for any type!

Terminology: e is *type correct* (in context Γ) if e has a type, i.e. $\Gamma \vdash e : \tau$ for some τ.

Type system for statements

Notation: $\Gamma \vdash s$ ("Statement s is type correct in context Γ")

Rules:

$$\Gamma \vdash skip$$

$$\frac{x : \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash s_1 ; s_2}$$

$$\frac{\Gamma \vdash b : \mathbf{bool} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash\ \mathbf{if}\ \mathrm{b}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2}$$

$$\frac{\Gamma \vdash b : \mathbf{bool} \qquad \Gamma \vdash s}{\Gamma \vdash \mathbf{while}\ b\ \mathbf{do}\ s}$$

Program $p = \Gamma; s$ is type correct iff. $\Gamma \vdash s$

Note: rules for $\Gamma \vdash e : \tau$ and $\Gamma \vdash s$ are a terminating algorithm (Prolog program!) for type checking.

3.2.2 Semantics for WHILE$_{\mathbf{T}}$

11/21/2001

$$\Sigma = Var \rightarrow Val, Val = T \cup \mathbb{Z}$$

where T is the set of truth values $\{$tt,ff$\}$

Assumption: T and $\mathbb{Z}$ are disjoint.
Need new value **err** $\notin Val$ to indicate a run time error.

$$Val_e = Val \cup \{\mathbf{err}\}$$

$\mathcal{A}[a_1\ aop\ a_2]\sigma = [aop](\mathcal{A}[a_1]\sigma, \mathcal{A}[a_2]\sigma)$

$$\begin{array}{rcl} [aop]_e : Val_e \times Val_e & \rightarrow & Val_e \\ [aop]_e(v_1, v_2) & = & \begin{cases} [aop](v_1, v_2) \text{ if } v_1, v_2 \in \mathbb{Z} \\ \mathbf{err} \text{ otherwise} \end{cases} \end{array}$$

Analogously:

$$[bop]_e : Val_e \times Val_e \rightarrow Val_e$$
$$[rop]_e : Val_e \times Val_e \rightarrow Val_e$$

$$\begin{array}{rcl} \mathcal{E} : Exp \rightarrow \Sigma & \rightarrow & Val_e \\ \mathcal{E}[n]\sigma & = & n \\ \mathcal{E}[x]\sigma & = & \sigma(x) \\ \mathcal{E}[\mathbf{true}\ /\mathbf{false}\ ]\sigma & = & \text{tt/ff} \\ \mathcal{E}[e_1\ aop\ e_2]\sigma & := & [aop]_e(\mathcal{E}[e_1]\sigma, \mathcal{E}[e_2]\sigma) \quad +\ bop,\ +\ rop \end{array}$$

Operational semantics for statements

Relation $Stm \times \Sigma \times \Sigma_e$ where $\Sigma_e = \Sigma \cup \{err\}$ $(err \notin \Sigma)$
Notation $\langle s, \sigma \rangle \rightarrow \sigma'$

Rules:

$$\begin{array}{rl} \langle \mathbf{skip}\ , \sigma \rangle \rightarrow \sigma & \\ \langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{E}[e]\sigma] & \text{if } \mathcal{E}[e]\sigma \neq \mathbf{err} \\ \langle x := e, \sigma \rangle \rightarrow \mathbf{err} & \text{if } \mathcal{E}[e]\sigma = \mathbf{err} \end{array}$$

Note: $I := 0$; $I := true$ is semantically ok, but not type correct!

$$\frac{\langle s_1, \sigma_1\rangle \rightarrow \sigma_2 \qquad \langle s_2, \sigma_2\rangle \rightarrow \sigma_3 \qquad (\sigma_2 \neq \textbf{err})}{\langle s_1; s_2, \sigma_1\rangle \rightarrow \sigma_3}$$

$$\frac{\langle s_1, \sigma_1\rangle \rightarrow \textbf{err}}{\langle s_1; s_2, \sigma_1\rangle \rightarrow \textbf{err}}$$

$$\frac{\mathcal{E}[e]\sigma \notin T}{\langle \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \sigma\rangle \rightarrow \textbf{err}}$$

$$\frac{\mathcal{E}[e]\sigma = \text{tt}/\text{ff} \qquad \langle s_1 \ / \ s_2, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle \textbf{if } e \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip }, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{while } e \textbf{ do } s, \sigma\rangle \rightarrow \sigma'}$$

Correctness of type system:
Type correct programs do not produce **err** (at runtime)

Completeness:
Programs that do not produce **err** are type correct.

$$(\textbf{if } b \textbf{ then } I := true \textbf{ else } I := 0; I := 1)$$

Correctness is desirable, completeness (in general) impossible.
Languages with a correct type system are called *type safe*:

- Examples: Java, ML, Haskell, Gofer, $C^\sharp$
- Counter-examples: Lisp, Eiffel, C, C++

3.2.3 WHILE$_T$ is type safe

Definition:

$$\begin{array}{rcl} typ & : & Val \rightarrow \{\textbf{int}, \textbf{bool}\} \\ typ(v) & = & \begin{cases} \textbf{int} & \text{if } v \in \mathbb{Z} \\ \textbf{bool} & \text{if } v \in T \end{cases} \end{array}$$

"σ is type correct with respect to Γ"
$\sigma : \Gamma \Leftrightarrow \forall x : t \in \Gamma. typ(\sigma(x)) = \tau$

Theorem: (Type safety)
$\Gamma \vdash s \wedge \sigma : \Gamma \wedge \langle s, \sigma\rangle \rightarrow \sigma' \Rightarrow \sigma' \neq \textbf{err} \wedge \sigma' : \Gamma$

Proof: by induction on $\langle s, \sigma\rangle \rightarrow \sigma'$

- Case $\langle x := e, \sigma\rangle \rightarrow \underbrace{\sigma[x \mapsto \mathcal{E}[e]\sigma]}_{\sigma'}, \ \mathcal{E}[e]\sigma \neq \textbf{err}$

$$\Gamma \vdash x := e \quad \Rightarrow \quad x : \tau \in \Gamma \wedge \Gamma \vdash e : \tau$$

$$\Rightarrow \quad typ(\mathcal{E}[e]\sigma) = t \ \text{(because } \sigma : \Gamma)$$

Lemma: $\Gamma \vdash e : \tau \wedge \sigma : \Gamma \Rightarrow typ(\mathcal{E}[e]\sigma) = \tau$

$$\begin{aligned} \sigma : \Gamma \quad &\Rightarrow \quad typ(\sigma(x)) = \tau = typ\left(\sigma'(x)\right) \\ &\Rightarrow \quad \sigma' : \Gamma \end{aligned}$$

3.3 A polymorphic type system

11/26/2001

Keywords: type variables, genericity

3.3.1 Type checking

Example: $x : \tau;\ y : \tau;\ x := y$ is type correct for any type τ

New syntax for types:

$$\tau ::= \textbf{int} \mid \textbf{bool} \mid \underbrace{\alpha \mid \beta \mid \gamma \mid \ldots}_{\text{type variable}}$$

Example: $x : \alpha;\ y : \alpha;\ x := y$

→ Semantics is unchanged because it is independent of the type language.

→ Type checking rules do not change either! Why? Because for type checking, type variables are treated like new atomic types.

$$\frac{\dfrac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \dfrac{y : \alpha \in \Gamma}{\Gamma \vdash y : \alpha}}{\underbrace{x : \alpha;\ y : \alpha}_{\Gamma} \vdash x := y}$$

Problem with type safety theorem: $\sigma : \Gamma$ only holds if Γ is monomorphic (does not contain type variables).

Idea: A (type variable) *substitution* is a function

$$\Theta : \ \text{type variables} \ \rightarrow \ \text{types}$$

Example:

$$\Theta = \{a \mapsto \textbf{bool}\,,\ \beta \mapsto \textbf{int}\,,\ \gamma \mapsto \delta, \ldots\}$$

Extension to types:

$$\begin{aligned} \Theta(\textbf{int}) \quad &= \quad \textbf{int} \\ \Theta(\textbf{bool}) \quad &= \quad \textbf{bool} \end{aligned}$$

Extension to contexts:

$$\Theta\Big(\{x_1 : \tau_1; \ldots\}\Big) = \{x : \Theta(\tau_1); \ldots\}$$

Theorem: $\Gamma \vdash s \wedge \sigma : \Theta(\Gamma) \wedge \langle s, \sigma\rangle \to \sigma' \Rightarrow \sigma' \neq \textbf{err} \wedge \sigma' : \Theta(\Gamma)$

Proof: similar to monomorphic case.
"If s is type correct with respect to Γ, then type safe execution is guaranteed for any state σ that is type correct with respect to an instance $\Theta(\Gamma)$ of Γ."

Example:

$$\{x : \alpha, y : \alpha\} \vdash s \Rightarrow \text{ start state } \begin{cases} \{x \mapsto 0, y \mapsto 1\} & \text{ok} \\ \{x \mapsto 0, y \mapsto \text{tt}\} & \text{not ok} \\ \{x \mapsto \text{tt}, y \mapsto \text{tt}\} & \text{ok} \end{cases}$$

3.3.2 Type inference / reconstruction

Compute missing declarations **Example:**

$$\begin{array}{rcl} \textbf{while } I > 0 \textbf{ do } I := I + 1 & \Rightarrow & I : \textbf{int} \\ \textbf{while } I > 0 \textbf{ do } I := I \wedge I & & \text{:-(} \\ x := y & \Rightarrow & x : \tau, y : \tau \text{ for any } \tau \end{array}$$

Key ideas:

1. Use type variable during reconstruction to stand for (yet) unknown types.
2. Use type checking rules for type reconstruction by interpreting the rules as a Prolog program, i.e. use rules backwards and instantiate type variables by unification.

$$\cfrac{\cfrac{\cfrac{(z:\gamma) \in \Gamma \quad \overset{\Rightarrow \beta=\gamma}{\Gamma \vdash y : \gamma}}{\Gamma \vdash z := y} \qquad \cfrac{(y:\beta) \in \Gamma \quad \overset{\Rightarrow \alpha=\beta}{\Gamma \vdash x : \beta}}{\Gamma \vdash y := x}}{\Gamma \vdash z := y;\ y := x} \qquad \cfrac{(x:\alpha) \in \Gamma \quad \overset{\Rightarrow \alpha=\textbf{int}}{\Gamma \vdash 0 : \alpha}}{\Gamma \vdash x := 0}}{\underbrace{x : \alpha, y : \beta, z : \gamma}_{\Gamma} \vdash (z := y;\ y := x);\ x := 0}$$

In Prolog:

$$\begin{array}{rcl} \text{Query} & : & \text{typecheck } ([x : \alpha, y : \beta, z : \gamma], z := y;\ y := x;\ x := 0) \\ \text{Result} & : & \alpha = \textbf{int} \\ & & \beta = \textbf{int} \\ & & \gamma = \textbf{int} \end{array}$$

Algorithm for type inference: Beginning with

$$\underbrace{\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}}_{\Gamma} \vdash s$$

apply the rules for $\Gamma \vdash s$ backwards while instantiating the α_i as much as necessary (unification!) $\Rightarrow$ We compute a substitution Θ such that $\{x_1 : \Theta(\alpha_1), \ldots, x_n : \Theta(\alpha_n)\} \vdash s$ is provable.

⇒ Type inference reduced to type checking

⇒ Type safety also guaranteed for type inference

Theorem: Let Θ be computed by the above algorithm. Then

1. $\Theta(\Gamma) \vdash s$
2. Θ is the "most general solution"
   If $\Theta'(\Gamma) \vdash s$ is provable, then Θ is more general than Θ' (Θ' is an *(*instance) of Θ):
   $\Theta' = \delta \circ \Theta$ for some substitution δ.

Example: For $x := y$, $\Gamma = \{x : \alpha, y : \beta\}$ the algorithm computes $\Theta = \{\beta \mapsto \alpha\}$
$\Theta' = \{\alpha \mapsto$ **bool** $, \beta \mapsto$ **bool** $\}$ is an instance of Θ: $\Theta' = \{\alpha \mapsto$ **bool** $\} \circ \Theta$

Classification of programming languages with respect to type systems

	monomorphic (no reuse)	**polymorphic**[7]
static	Pascal, Modula, (Fortran)	ML, Haskell, Gofer, C, Java, $C^\sharp$
dynamic	not sensible	LISP

[7] allows generic (with respect to types) programs

4 Denotational Semantics

11/28/2001

1. The semantics of inductive definitions (including rule induction)
2. A denotational semantics of While based on sets
3. Fixpoint theory
4. A direct denotational semantics of While
5. Extensions of While

4.1 Inductive Definitons

Rules:

$$\frac{a_1 \in X \qquad \dots \qquad a_n \in X}{a \in X}$$

defines $X \subseteq A$ where X is the set to be defined and $a_1, \dots, a_n, a \in A$. In the sequel: $R \subseteq \wp(A) \times A$ is a rule.

Note: R, X is potentially infinite.

Definition: $B \subseteq A$ is *R-closed* iff. for all $(H, a) \in R$:

$$H \subseteq B \Rightarrow a \in B$$

Definition: I_R is the least R-closed subset of A.

Note: There is always a least R-closed subset.

$$B_1, B_2 \text{ R-closed } \Rightarrow B_1 \cap B_2 \text{ R-closed}$$

$I_R = \bigcap\{B \subseteq A \mid \text{ B is R-closed } \}$

Why the least set? No junk! I_R should contain only those elements that R forces to be in it.

Example: $A = \mathbb{R}$, Rules: $\frac{}{0 \in X}$, $\frac{n \in X}{n+1 \in X}$

Least R-closed subset of $\mathbb{R}$: $\mathbb{N}$. Also R-closed: $\mathbb{R}, \mathbb{Q}, \mathbb{Z}$

Set theoretic way of writing $\mathbb{R}$:

$$\{(\emptyset, 0)\} \cup \{(\{n\}, n+1) \mid n \in \mathbb{R}\}$$

4.1.1 Rule induction

Aim:

$$\frac{\forall (\{h_1, \dots, h_n\}, a) \in R . P(h_1) \wedge \dots \wedge P(h_n) \Rightarrow P(a)}{\forall x \in I_R . P(x)}$$

$\langle s, \sigma \rangle \rightarrow \sigma'$
$(s, \sigma, \sigma') \in OpSem$

$$I_R \subseteq P \Rightarrow I_R = I_R \cap P \Rightarrow I_R \cap P \text{ is R-closed} \overset{\text{leastness of } I_R}{\Longrightarrow} I_R \subseteq I_R \cap P \Rightarrow I_R \subseteq P$$

$$\forall x \in I_R.P(x)) \quad \Leftrightarrow \quad (\forall (H,a) \in R.\underbrace{H \subseteq I_R}_{\text{additional assumption}} \wedge (\forall h \in H.P(h)) \Rightarrow \underbrace{a \in I_R}_{\text{tt}} \wedge P(a))$$

$$\overset{\text{Rule induction}}{\Leftarrow}$$

Generalized rule induction:

$$\frac{\forall (H,a) \in R.H \subseteq I_R \wedge (\forall h \in H.P(h)) \Rightarrow P(a)}{\forall x \in I_R.P(x)}$$

Side conditions

Example:

$$\frac{B[b]\sigma = \text{ff}}{\langle w, \sigma \rangle \rightarrow \sigma'}$$

Need to distinguish two kinds of premises:

a_i and everything else ("side conditions")

$$\frac{a_1 \in X, \ldots, a_n \in X \qquad C_1 \ldots C_m}{a \in X} \tag{4}$$

With side conditions:

$$\frac{\forall \text{ rules } (4)\ .a_1, \ldots, a_n \in I_R \wedge P(a_1) \wedge \ldots \wedge P(a_n) \wedge C_1 \wedge \ldots \wedge C_m \Rightarrow P(a)}{\forall x \in I_R.P(x)}$$

4.1.2 I_R as a least fixed point ($f(x) = x$)

Assumption: R is *finitary*: $(H,a) \in R \Rightarrow H$ finite

$\hat{R} : \wp(A) \rightarrow \wp(A)$
$\hat{R}(B) = \{a \mid \exists (H,a) \in R.H \subseteq B\}$ "one step consequence operator"

To show:

1. I_R is least fixpoint of $\hat{R}$
2. $I_R = \bigcup_{i \in \mathbb{N}} \hat{R}^i(\emptyset)$

Easy to see:

1. B is R-closed iff. $\hat{R}(B) \subseteq B$
2. $\hat{R}$ is monotone ($B_1 \subseteq B_2 \Rightarrow \hat{R}(B_1) \subseteq \hat{R}(B_2)$)

$\hat{R}$ generates/approximates I_R:

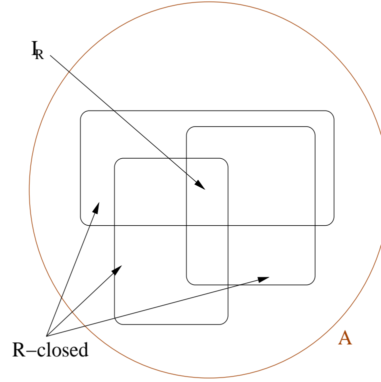
$$\begin{aligned} A_0 &:= \hat{R}^0(\emptyset) = \emptyset \text{ - at the start we know nothing} \\ A_1 &:= \hat{R}^1(\emptyset) = \hat{R}(\emptyset) \text{ - all axioms/facts} \\ A_{n+1} &:= \hat{R}^{n+1}(\emptyset) = \hat{R}(\hat{R}^n(\emptyset)) = \hat{R}(A_n) \\ &\quad \text{- all elements of } A \text{ that can be derived with a derivation tree of depth } n+1 \end{aligned}$$

$A_\omega := \bigcup_{i\in\mathbb{N}} A_i$

Example: $A = \mathbb{N}$ Rules: $\frac{}{0 \in X}$, $\frac{n \in X}{n+2 \in X}$

$$\begin{array}{rcl} \hat{R}(M) & = & \{0\} \cup \{m+2 \mid m \in M\} \\ \hat{R}^0(\emptyset) & = & \emptyset \\ \hat{R}^1(\emptyset) & = & \{0\} \\ \hat{R}^2(\emptyset) & = & \{0,2\} \\ \hat{R}^3(\emptyset) & = & \{0,2,4\} \\ \hat{R}^n(\emptyset) & = & \{0,\ldots,2(n-1)\} \end{array}$$

$\bigcup_{i\in\mathbb{N}} \hat{R}^i(\emptyset) =$ set of even numbers



The $(A_i)_{i\in\omega}$ forms an ω-chain:

$$A_0 \subseteq A_1 \subseteq A_2 \subseteq \ldots \subseteq A_n \subseteq \ldots$$

Proof: $A_i \subseteq A_{i+1}$ by induction on i

$$\begin{array}{rcl} i=0 & : & A_0 = \emptyset \subseteq A_1 \\ i \to i+1 & : & A_i \subseteq A_{i+1} \stackrel{?}{\Rightarrow} A_{i+1} \subseteq A_{i+2} \\ & & \hat{R} \text{ monotone: } A \subseteq B \Rightarrow \hat{R}(A) \subseteq \hat{R}(B) \\ & & \hat{R}(A_i) \subseteq \hat{R}(A_{i+1}) \end{array}$$

Theorem: If R is finitary

- A_ω is R-closed
- $\hat{R}(A_\omega) = A_\omega$
- $A_\omega = I_R$

Proof:

- Let $(\{h_1,\ldots,h_n\},a) \in R$, $\{h_1,\ldots,h_n\} \subseteq A_\omega$
  Show $a \in A_\omega$:
  $\forall 1 \leq i \leq n. \exists k_i. h_i \in A_{k_i}$
  $k := max\{k_1,\ldots,k_n\} \Rightarrow \forall 1 \leq i \leq n. k_i \in A_k \Rightarrow a \in A_{k+1} \Rightarrow a \in A_\omega$

- $\hat{R}(A_\omega) \subseteq A_\omega \wedge A_\omega \subseteq \hat{R}(A_\omega)$

  Let $a \in A_\omega$; show $a \in \hat{R}(A_\omega)$
  $a \in A_\omega \Rightarrow \exists n.a \in A_n \Rightarrow a \in \hat{R}(A_n) \subseteq \hat{R}(A_\omega)$
  because $A_n \subseteq A_\omega$ and $\hat{R}$ monotone

- Show A_ω is the least R-closed set. Let B be an R-closed set ($\hat{R}(B) \subseteq B$).
  Show $A_\omega \subseteq B : \forall i.A_i \subseteq B$ by induction on i.

$$\begin{aligned} i = 0 \quad &: \quad A_0 \subseteq B \ (A_0 = \emptyset) \\ i \to i+1 \quad &: \quad A_i \subseteq B \overset{?}{\Rightarrow} A_{i+1} \subseteq B = \hat{R}(A_i) \subseteq \hat{R}(B) \end{aligned}$$

Corollary:

1. $I_R = \bigcup_{i \in \omega} \hat{R}^i(\emptyset)$
2. I_R is least fixpoint of $\hat{R}$

Proof: for 2:
Show: every fixpoint of $\hat{R}$ is R-closed:

$$\hat{R}(B) = B \Rightarrow \hat{R}(B) \subseteq B \Rightarrow B \text{ is R-closed}$$

Notation: $A_\omega = fix(\hat{R})$

4.2 Denotational Semantics of WHILE

- operational: close to implementation
- denotational: more abstract, easier to use in proofs

Idea: For every syntactic class C (Stmt, Aexp, ...) there is one function $D_C : C \to M$, where M is a semantic domain, e.g. a known mathematical structure.

- $A : Aexp \to (\Sigma \to \mathbb{N})$
- $S : Stmt \to (\Sigma \to \Sigma)$ (operation sem.: $Stmt \times \Sigma \times \Sigma$)
  - $S[skip] = Id$ $\quad$ ($S[skip]\sigma = \sigma$)
  - $S[x := a]\sigma = \sigma[x \mapsto A[a]\sigma]$
  - $S[s_1; s_2] = S[s_2] \circ S[s_1]$ $\quad$ ($S[s_1; s_2]\sigma = S[s_2](S[s_1]\sigma)$)
  - S[if b then s_1 else s_2]$\sigma =$
    - $S[s_1]\sigma$ if $B[b]\sigma = $ tt
    - $S[s_2]\sigma$ if $B[b]\sigma = $ ff

– a try for while:

$$S[\text{while b do s}]\sigma = \begin{cases} \sigma & \text{if } B[b] = \text{ff} \\ S[w](S[s]\sigma) & \text{if } B[b] = \text{tt} \end{cases} \tag{5}$$

Ω = while true do skip
$S[\Omega] = S[\Omega] \circ S[skip]$[8] $= S[\Omega]$:-(

Wanted: least function satisfying (5) (here: "least" means least defined function)
Why? Termination only if explicitly required by the definition ("no garbage").
Equivalence to operational semantics.

General principle for recursive definitions / solutions to recursive equations

Recursive:

$$f = \underbrace{\ldots f \ldots}_{e} \quad f \in A \text{ (e.g. } A = B \to C) \tag{6}$$

Wanted: *Least* solution f. Define $F(f) := e$ [9]
Now if f is the solution to (6) $\Rightarrow F(f) = f$

Wanted: $fix(F) =: f$, then f is by definition the least solution of (6).

Problem: When does $fix(F)$ exist?
First solution for while: define $\Sigma \to \Sigma$ as set

1. Transform $S[w]\sigma$ into a recursive equation for $S[w]$, treating functions ($S[w]\sigma$) as relations ($S[w]$).

$$S[w] = \{(\sigma,\sigma) \mid B[b]\sigma = \text{ff}\} \cup \underbrace{\{(\sigma, S[w](S[s]\sigma)) \mid B[b]\sigma = \text{tt} \wedge \exists \sigma'.(\sigma,\sigma') \in S[s] \wedge (\sigma',\sigma'') \in S[w]\}}_{\Gamma(S[w])}$$

Wanted: $fix(\Gamma)$
$\Gamma : (\Sigma \nrightarrow \Sigma) \to (\Sigma \nrightarrow \Sigma)$
Intuition: if $\Phi : \Sigma \nrightarrow \Sigma$ is the meaning of some $s' \in Stm$ then $\Gamma(\Phi)$ is the meaning of **if** b **then** $s;s'$ **else skip**
$\Rightarrow \Gamma$ unfolds the loop once.
if b **then** $\{s$; **if** b **then** $s;\ldots$ **else skip** $\}$ **else skip** $\}$ infinite unfolding

12/05/2001

2. Find set of rules R such that $\hat{R} = \Gamma$.

$$R = \{(\emptyset, (\sigma,\sigma)) \mid B[b]\sigma = \text{ff}\} \cup \{(\{(\sigma',\sigma'')\}, (\sigma,\sigma'')) \mid B[b]\sigma = \text{tt} \wedge (\sigma,\sigma') \in S[s]\}$$

Fact: $\hat{R} = \Gamma$

Definition: $S[w] = fix(\Gamma) = \bigcup_{i\in\mathbb{N}} \hat{R}^i(\emptyset)$ $[N.B.\ S[w] \in \Sigma \nrightarrow \Sigma]$

Intuition: $\hat{R}^i(\emptyset)$ = semantics of the first i iterations

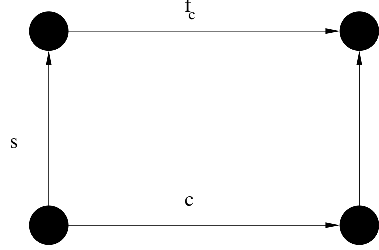
Denotational semantics is *compositional*:

$$S[c(t_1,\ldots,t_n)] = f_c(S[t_1],\ldots,S[t_n])$$

c is composition of statements

[8] Id
[9] $F \in A \to A$

Recursion theory: primitive recursion
Mathematics: Homomorphism (from syntax to semantics)



Theorem: $S[w] = S[$**if** b **then** $s; w$ **else skip** $]$

Proof:

$$\begin{array}{rcl} S[w]\sigma & = & fix(\Gamma)\sigma = \Gamma(fix(\Gamma))\sigma = \Gamma(S[w])\sigma \\ & = & \begin{cases} \sigma \;\; if B[b]\sigma = \text{ff} \\ S[w](\underbrace{S[s]\sigma}_{\sigma'}) \;\; if B[b]\sigma = \text{tt} \end{cases} \\ & = & \begin{cases} S[\textbf{skip}]\sigma \;\; if B[b]\sigma = \text{ff} \\ S[s;w]\sigma \;\; if\ B[b]\sigma = \text{tt} \end{cases} \\ & = & S[\textbf{if}\ \text{b}\ \textbf{then}\ \text{s;w}\ \textbf{else skip}]\sigma \end{array}$$

Equivalence of operational & denotational semantics

Aim: $\langle s, \sigma \rangle \to \sigma' \Leftrightarrow S[s]\sigma = \sigma'$
Theorem: $\langle s, \sigma \rangle \to \sigma' \Rightarrow S[s]\sigma = \sigma'$

Proof: by induction on $\langle s, \sigma \rangle \to \sigma'$

$$\frac{\langle s, \sigma \rangle \to \sigma' \quad \langle w, \sigma' \rangle \to \sigma''}{\langle w, \sigma \rangle \to \sigma''} B[b]\sigma = \text{tt}$$

$$\begin{array}{rcl} S[w]\sigma & = & \Gamma(S[w])\sigma \\ & \stackrel{\Gamma}{=} & S[w](S[s]\sigma) \\ & \stackrel{I.H.1}{=} & S[w]\sigma' \\ & \stackrel{I.H.2}{=} & \sigma'' \\ & & \text{(Leastness not required)} \end{array}$$

Theorem: $S[s]\sigma = \sigma' \Rightarrow \langle s, \sigma \rangle \to \sigma'$

Proof: by induction on s

- Case $s = s_1; s_2$
  $S[s]\sigma = S[s_2](S[s_1]\sigma)$

$\Rightarrow \exists \sigma_1 . S[s_1]\sigma = \sigma_1$
$\overset{I.H.}{\Rightarrow} \langle s_1, \sigma \rangle \rightarrow \sigma_1$
$\overset{I.H.}{\Rightarrow} \langle s_2, \sigma \rangle \rightarrow \sigma'$
$\Rightarrow \langle s_1; s_2, \sigma \rangle \rightarrow \sigma'$

- $s = \textbf{while } b \textbf{ do } s_0$
  $S[s]\sigma = \sigma'$
  $\Rightarrow (\sigma, \sigma') \in S[s] = fix(\Gamma) = \bigcup_{n \in \mathbb{N}} \hat{R}^n(\emptyset)$
  $\Rightarrow \exists n . (\sigma, \sigma') \in \hat{R}^n(\emptyset)$
  $\Rightarrow \langle s, \sigma \rangle \rightarrow \sigma'$
  by auxiliary lemma
  $R = \{(\emptyset, (\sigma, \sigma)) \mid B[b]\sigma = \text{ff}\} \cup \{(\{(\sigma', \sigma'')\}, (\sigma, \sigma'')) \mid B[b]\sigma = \text{tt} \wedge (\sigma, \sigma') \in S[s_0]\}$

  Lemma: (local)
  $\forall \sigma, \sigma' . (\sigma, \sigma') \in \hat{R}^n(\emptyset) \Rightarrow \langle s, \sigma \rangle \rightarrow \sigma'$

  Proof: by induction on n

  - $n = 0 : \checkmark$
  - $n \rightarrow n+1 : (\sigma, \sigma') \in \hat{R}^{n+1}(\emptyset) = \hat{R}(\hat{R}^n(\emptyset))$
    Case distinction:
    * $B[b]\sigma = \text{ff}\checkmark$
    * $B[b]\sigma = \text{tt} : \exists \sigma'' . (\sigma, \sigma'') \in S[s_0] \wedge (\sigma'', \sigma') \in \hat{R}^n(\emptyset)$
      $\Rightarrow \underbrace{\langle s_0, \sigma \rangle \rightarrow \sigma''}_{\text{induction on s}} \wedge \langle s, \sigma'' \rangle \rightarrow \sigma'$
      $\Rightarrow \langle s, \sigma \rangle \rightarrow \sigma'$ qed.

Theorem:

$$\langle s, \sigma \rangle \rightarrow \sigma' \Leftrightarrow S[s]\sigma = \sigma'$$

4.3 Complete partial orders (cpos)[10]

12/10/2001

Generalization of $\wp A, \hat{R} : \wp A \rightarrow \wp A \quad fix(\hat{R}) = \bigcup \ldots$
Definition: Let $(P, \sqsubseteq)$ be a partial order, $X \subseteq P, u \in P$.

- u is an *upper bound* of X iff. $\forall x \in X . x \sqsubseteq u$
- u is *least upper bound*[11] of X iff. u is an upper bound of X and if u' is an upper bound of X then $u \sqsubseteq u'$

[10] "Vollständige Halbordnungen"
[11] least upper bound = "Supremum"

Remark: Least upper bounds don't need to exist ($\underbrace{..}_{X}$), but if they exist, they are unique: u_1, u_2 are least upper bounds of X then $u_2 \sqsubseteq u_1$ and $u_1 \sqsubseteq u_2$ and therefore $u_1 = u_2$.

Notation: If X has a least upper bound, it is written $\bigsqcup X$
$x_1 \sqcup x_2 := \bigsqcup\{x_1, x_2\}$. NB: $x \sqsubseteq y \Rightarrow x \sqcup y = y$

Definition: Let $(D, \sqsubseteq)$ be a partial order (D = "Domain"). $(D, \sqsubseteq)$ is a *complete partial order* (cpo) iff. all ω-chains $K \subseteq D$ have a least upper bound $\bigsqcup K$.
An ω-chain is a set $\{d_0, d_1, \ldots\}$ such that $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$
$(D, \sqsubseteq)$ is a *cpo with* $\perp$ if $\perp \in D$ is the least element of D.

Example:
$\left(P(A), \subseteq\right)$ is a cpo with $\perp = \emptyset$ and $\bigsqcup = \bigcup$

$(A \nrightarrow A, \sqsubseteq)$
$f := (\bigcup_{i\in\mathbb{N}} f_i) \in A \nrightarrow A$ if each $f_i \in A \nrightarrow A$ and if $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$

Proof: $(a_1, a_2), (a_1, a_3) \in f$
Without loss of generality $(a_1, a_2) \in f_i, (a_1, a_3) \in f_j, i \le j$
$\Rightarrow f_i \subseteq f_j \Rightarrow (a_1, a_2) \in f_j \in A \nrightarrow A$
$\Rightarrow a_2 = a_3$

Generalization:

$$\begin{array}{rcl} \wp A & \rightsquigarrow & D \\ \subseteq & \rightsquigarrow & \sqsubseteq \\ \emptyset & \rightsquigarrow & \perp \\ \hat{R} & \rightsquigarrow & \text{continuous function} \end{array}$$

Why does $\hat{R}$ have a least fixpoint? Because it is *monotone*.
Why $fix(\hat{R}) = \bigcup_{i\in\mathbb{N}} \hat{R}^i(\emptyset)$? Because $\hat{R}$ is *continuous*: $\hat{R}(\bigcup_{i\in\mathbb{N}} B_i) = \bigcup_{i\in\mathbb{N}} \hat{R}(B_i)$ where $B_0 \subseteq B_1 \subseteq \ldots$

Counter-example to continuity:

$$\frac{}{0} \quad \frac{n}{n+2} \quad \frac{0 \quad 2 \quad 4 \quad \ldots}{1}$$

$B_i = \{0, 2, 4, \ldots, 2(i-1)\}$
$B_0 = \emptyset, B_1 = \{0\}, B_2 = \{0, 2\}, \ldots$

$$\begin{array}{c} \hat{R}(\bigcup_i B_i) = \hat{R}(\{0,2,4,\ldots\}) = \{0,2,4,\ldots\} \cup \{1\} \\ \neq \\ \bigcup_i \hat{R}(B_i) = \bigcup_i B_{i+1} = \{0,2,4,\ldots\} \end{array}$$

Definition:
Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be cpos, and let $f : D \rightarrow E$.
f is monotone iff. $d_1 \sqsubseteq_D d_2 \Rightarrow f(d_1) \sqsubseteq_E f(d_2)$
f is continuous iff. f is monotone and for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ we have

$$\underbrace{f(\bigsqcup_i d_i)}_{\text{exists because } D \text{ cpo}} = \underbrace{\bigsqcup_i f(d_i)}_{\text{exists because } E \text{ cpo}}$$

Theorem: (Knaster-Tarski Fixpoint theorem)
Let $(D, \sqsubseteq)$ be a cpo with $\bot$ and $f : D \to D$ be continuous[12]. Then $fix(f) = \bigsqcup f^i(\bot)$ is the least fixpoint of f.

Proof:
$\bigsqcup_i f^i(\bot)$ exists because $\{f^i(\bot)\}_{i\in\mathbb{N}}$ is an ω-chain:
(Proof by induction on $i : f^i(\bot) \sqsubseteq f^{i+1}(\bot)$)

$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot))\dots$

(i) $f(fix(f)) = fix(f)$:

$$\begin{aligned} f(\bigsqcup_i f^i(\bot)) &= \bigsqcup_i f^{i+1}(\bot) \\ &= \bot \sqcup \bigsqcup_i f^{i+1}(\bot) \\ &= f^0(\bot) \sqcup \bigsqcup_i f^{i+1}(\bot) = \bigsqcup_i f^i(\bot) \\ &\overset{def}{=} fix(f) \end{aligned}$$

$(f^0(x) = x)$

(ii) $f(p) \sqsubseteq p \Rightarrow fix(f) \sqsubseteq p$
By induction on $i : f^i(\bot) \sqsubseteq p$

(a) $f^0(\bot) = \bot \sqsubseteq p$

(b) $f^{i+1}(\bot) = \underbrace{f(f^i(\bot)) \sqsubseteq f(p)}_{\text{I.H. + mono}(f)} \sqsubseteq p$

$\Rightarrow \bigsqcup_i f^i(\bot) \sqsubseteq p$ because p is an upper bound of $\{f^i(\bot)\}_{i\in\mathbb{N}}$ and $\bigsqcup$ is the least upper bound.

Dana Scott: "Domain theory"

Thesis: Any computable function is continuous with respect to a suitable ordering.

$$f(x) = \textbf{if } x \leq 0 \textbf{ then } 0 \textbf{ else } 1$$

not continuous in the sense of analysis.

Application to **WHILE** :

1. $\Sigma \rightharpoonup \Sigma$ is a cpo
2. For **while** : $\Gamma : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ has a least fixpoint because:
   a) $\Gamma = \hat{R}$ for a finitary R. ✓
   b) Γ is continuous – tedious[13] if proved directly. Better: build general theory of continuous functions, show that Γ is a composition of continuous functions.

[12] "stetig"
[13] "mühsam", "langweilig, "nervtötend"

λ-Notation

$$\begin{array}{rcl} f(x) = x+1 & \leadsto & f = \lambda x.x+1 \\ f(5) = 5+1 & \leadsto & f\ 5 = (\lambda x.x+1)5 \rightarrow (x+1)[5/x] = 5+1 \end{array}$$

$\lambda x.t$: x parameter, t result

Typing rules:

$$\frac{x : D \qquad t : E}{\lambda x.t : D \rightarrow E} \qquad \frac{f : D \rightarrow E \qquad t : D}{f\ t : E}$$

α-Conversion[14]: $\lambda x.x+1 = \lambda y.y+1$

$$\begin{array}{rcl} \lambda x.x+y & \neq & \lambda x.x+z \\ \lambda x.\lambda y.x+y & \neq & \lambda x.\lambda x.x+x \end{array}$$

12/12/2001

$$\beta : (\lambda x.s)t \quad =_\beta \quad s[t/x] \text{ (Substitution of } t \text{ for } x \text{ in } s\text{)}$$

$$\begin{array}{rcl} (\lambda x.\lambda y.x+y)y & \neq & \lambda y.y+y \\ = (\lambda x.\lambda y'.x+y')y & & \\ & = & \lambda y'.y+y' \end{array}$$

⇒ Definition of $s[t/x]$ needs to rename bound variables to avoid name clashes.

$$\eta : \lambda x.(t\ x) =_\eta t \quad \text{if } x \text{ does not occur free in } t$$

Example:

$$\begin{array}{rcl} \lambda x.\sin x & = & \sin \\ \lambda x.(x\ x) & \neq & x \end{array}$$

Construction of cpos

Aim: meta-language for defining semantics.

4.3.1 Discrete cpos

Definition: $\sqsubseteq$ is a *discrete* partial order iff. $\forall x,y.x \sqsubseteq y \Leftrightarrow x = y$

Fact: Every discrete partial order is a cpo.

Fact: If D is discrete, and $f : D \rightarrow E$, then f is continuous.

[14] "Gebundene Namen sind Schall und Rauch."

4.3.2 Cartesian Products

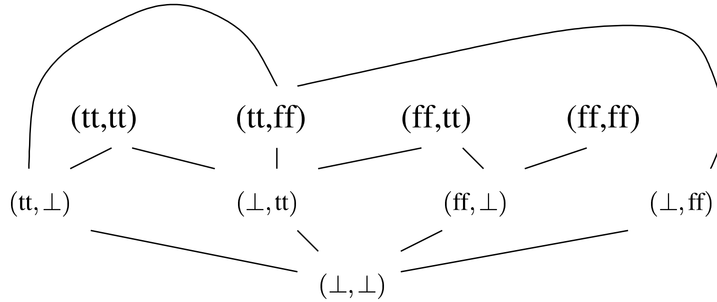
$D := D_1 \times \ldots \times D_n$ (D_i are cpos)
$(d_1, \ldots, d_n) \sqsubseteq_D (d'_1, \ldots, d'_n) \Leftrightarrow_{def} (d_1 \sqsubseteq_{D_1} d'_1) \wedge \ldots \wedge (d'_n \sqsubseteq_{D_n} d'_n)$
$(D, \sqsubseteq_D)$ is cpo:

$$\bigsqcup_{i \in \mathbb{N}} (d_1^i, \ldots, d_n^i) := (\bigsqcup_{i \in \mathbb{N}} d_1^i, \ldots, \bigsqcup_{i \in \mathbb{N}} d_n^i) \tag{7}$$

Need to check that right hand side of (7) exists and is least upper bound of left hand side.

Example:
$T_\perp := \frac{\text{tt} \quad \text{ff}}{\perp}$ (means: $\perp \sqsubseteq$ tt, $\perp \sqsubseteq$ ff)

$T_\perp \times T_\perp$:



The empty Cartesian product $\mathbf{1} = \{()\}$????

Lemma: Let E be a cpo, $e_{mn} \in E$ for $m, n \in \mathbb{N}$ such that $m \leq m' \wedge n \leq n' \Rightarrow e_{mn} \sqsubseteq_E e_{m'n'}$.
Then $\bigsqcup_i \bigsqcup_j e_{ij} = \bigsqcup_{i,j} e_{ij} = \bigsqcup_n e_{nn}$

$$\begin{array}{ccccc} \vdots & & & \iddots & \\ e_{01} & \sqsubseteq & e_{11} & \sqsubseteq & \ldots \\ & \rotatebox{45}{$\sqsubseteq$} & & & \\ e_{00} & \sqsubseteq & e_{10} & \sqsubseteq & \ldots \end{array}$$

Theorem: $f : \underbrace{D_1 \times \ldots \times D_n}_{D} \to E$ is continuous iff. it is continuous in each argument:
for all $1 \leq i \leq n$ and all $(d_1, \ldots, d_n) \in D$

$$f_i(x) := f(d_1, \ldots, d_{i-1}, x, d_{i+1}, \ldots, d_n) \text{ is continuous}$$

Proof:
$\Rightarrow \checkmark$
$\Leftarrow$: (only case $n = 2$)

1. f is monotone: $(x_0, y_0) \sqsubseteq (x_1, y_1) \Leftrightarrow$

$$f(x_0, y_0) \overset{f_1 \text{ monotone}}{\sqsubseteq} f(x_1, y_0) \overset{f_2 \text{ monotone}}{\sqsubseteq} f(x_1, y_1)$$

2. $(x_0, y_0) \sqsubseteq (x_1, y_1) \sqsubseteq \dots$

$$\begin{aligned} f\left(\bigsqcup_i (x_i, y_i)\right) &= f\left(\bigsqcup_i x_i, \bigsqcup_j y_j\right) \\ &= \bigsqcup_i f\left(x_i, \bigsqcup_j y_j\right) \quad f_1 \text{ is continuous} \\ &= \bigsqcup_i \bigsqcup_j f(x_i, y_j) \quad f_2 \text{ is continuous} \\ &= \bigsqcup_i f(x_i, y_i) \quad \text{qed.} \end{aligned}$$

4.3.3 Function spaces

Let D, E be cpos.
$[D \to E] :=$ set of continuous functions from D to E.
$f \sqsubseteq_{[D \to E]} g \Leftrightarrow_{def} \forall d \in D. f(d) \sqsubseteq_E g(d)$ "pointwise ordering" **Fact:** $\sqsubseteq_{[D \to E]}$ is a cpo.

If $\perp$ is least element of E, then $\lambda d.\perp$ is least element of $[D \to E]$.
Let $f_0 \sqsubseteq f_1 \sqsubseteq \dots$ be an ω-chain in $[D \to E]$.

$$\left(\begin{aligned} &\Rightarrow \quad f_0(d) \sqsubseteq f_1(d) \sqsubseteq \dots \\ &\Rightarrow \quad \bigsqcup_i f_i(d) \text{ exists} \end{aligned} \right)$$

Then the f_i have a supremum:

$$\bigsqcup_i f_i = \lambda d. \bigsqcup_i f_i(d)$$

Need to check that right hand side is supremum and right hand side is continuous.

4.3.4 Lifting

$$D \leadsto D_\perp$$

Let D be a set. $D_\perp := \{\perp\} \cup \{\lfloor d \rfloor \mid d \in D\}$ (disjoint union)
$\lfloor . \rfloor : D \to D_\perp$ "Boxing / Tagging"
such that $\lfloor d \rfloor \neq \perp$
$\lfloor d_1 \rfloor = \lfloor d_2 \rfloor \Rightarrow d_1 = d_2$

Theorem: $\lfloor . \rfloor$ always exists ("by construction").

$T_\perp = \{\lfloor \text{tt} \rfloor, \lfloor \text{ff} \rfloor, \perp\}$

If D is a (c)po then $D_\perp$ is a (c)po:

$$d_1' \sqsubseteq_{D_\perp} d_2' \Leftrightarrow_{def} (d_1' = \perp) \vee \exists d_1, d_2 \in D. d_1' = \lfloor d_1 \rfloor \wedge d_2' = \lfloor d_2 \rfloor \wedge d_1 \sqsubseteq_D d_2$$

If D is discrete, then $D_\perp$ is called a *flat* cpo. $x \sqsubseteq y \Leftrightarrow x = \perp \vee x = y$

$$\begin{array}{c} (\;.\;.\;.\;.\;.\;.\;) \\ \perp \end{array}$$

Remark: $\lfloor . \rfloor$ is continuous (if D is a cpo)

Strict extension of functions from D to $D_\perp$: (**Definition:** f is strict iff. $f(\perp) = \perp$)

$f : [D \to E], D$ cpo, E cpo with $\perp$
$f^* : [D_\perp \to E]$
$f^*(d') = \begin{cases} \perp & \text{if } d' = \perp \\ f(d) & \text{if } d' = \lfloor d \rfloor \end{cases}$
f^* is the strict extension of f.

Notation: x variable, d, e are terms. $e \in E$, $d \in D_\perp$

let $x \Leftarrow d.e \equiv (\lambda x.e)^* d$
let $x_1 \Leftarrow d_1, \ldots, x_n \Leftarrow d_n.e \equiv$ **let** $x_1 \Leftarrow d_1 \ldots$ **let** $x_n \Leftarrow d_n.e$

Example:

$$\begin{array}{rcl} \textbf{let } x \Rightarrow \lfloor 5 \rfloor . x+2 & = & (\lambda x.x+2)^* \lfloor 5 \rfloor \\ & = & (\lambda x.x+2) 5 = 7 \\ \perp & = & \perp_{D_\perp} = \perp_E \end{array}$$

Given $\otimes : S \times S \to S$
define $\otimes_\perp : S_\perp \times S_\perp \to S_\perp$
$s_1' \otimes s_2' =$ **let** $x_1 \Rightarrow s_1', x_2 \Rightarrow s_2'. x_1 \otimes x_2$

Miscellaneous functions

$cond : T \times D \times D \to D, D$ cpo
$cond(\text{tt}, x, y) = x$
$cond(\text{ff}, x, y) = y$
is continuous.

Proof:
First argument: T discrete
Second argument: $\lambda x.cond(b, x, y)$ is either $\lambda x.x$ (if $b = \text{tt}$) or $\lambda x.y$ – both are continuous.

update: $\Sigma \times Var \times \mathbb{Z} \to \Sigma$
$\sigma[x \mapsto n]$ is continuous because $\Sigma \times Var \times \mathbb{Z}$ is discrete.

12/17/2001

fix: $\big[[D \to D] \to D\big]$ (continuity of fix needs proof!) [WINSKEL93]

Theorem: (Winskel)
Well-typed λ-terms containing only continuous functions are continuous (in their free variables).

⇒ No more worry about continuity, as long as basic functions are continuous.

⇒ Every recursion equation $x = \Gamma(x)$ has a least solution $x = fix(\Gamma)$

4.4 A cpo-based denotational semantics for WHILE

$\Sigma \to \Sigma_\perp \cong \Sigma \nrightarrow \Sigma$

$$\begin{array}{rcl}
D : Stm & \to & (\Sigma \to \Sigma_\perp) \\
D[\,\textbf{skip}\,] & = & \lambda\sigma.\lfloor\sigma\rfloor \\
D[x := a] & = & \lambda\sigma.\lfloor\sigma[x \mapsto A[a]\sigma]\rfloor \quad A \text{ continuous because } Aexp \text{ and } \Sigma \text{ are discrete} \\
D[s_1; s_2] & = & \big(\lambda\sigma.\textbf{let } \sigma' = D[s_1]\sigma^{15}.D[s_2]\sigma'\big) = D[s_2] \circ_\perp D[s_1]^{9} \\
D[\,\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2] & = & \lambda\sigma.cond(B[b]\sigma, D[s_1]\sigma, D[s_2]\sigma\,) \\
D[\,\textbf{while } b \textbf{ do } s\,] & = & \text{least solution of} \\
 & & \Phi = \lambda\sigma.cond\big(B[b]\sigma, (\Phi \circ_\perp D[s])\sigma, \lfloor\sigma\rfloor\big) \\
 & = & fix\underbrace{\big(\lambda\Phi.\lambda\sigma.cond\,(B[b]\sigma, (\Phi \circ_\perp D[s])\sigma, \lfloor\sigma\rfloor)\big)}_{\Gamma}
\end{array}$$

Example:

$s = \textbf{while } i \neq 0 \textbf{ do } i := i - 1$

$$\begin{array}{rcl}
D[s]\{i \mapsto 2\} & = & fix(\Gamma)\{i \mapsto 2\} \\
 & = & \Gamma(fix(\Gamma))\{i \mapsto 2\} \quad \text{fix unfolded once} \\
 & = & \Big(\lambda\sigma.cond\big(\,B[i \neq 0]\{i \mapsto 2\}, fix(\Gamma) \circ_\perp D[i := i-1]\,\big)\{i \mapsto 2\}, \quad \underbrace{\lfloor\{i \mapsto 2\}\rfloor}_{\text{argument for } \lambda\sigma....}\Big) \\
 & = & \textbf{let } \sigma' = D[i := i-1]\{i \mapsto 2\}.fix(\Gamma)\sigma' \\
 & = & fix(\Gamma)\{i \mapsto 1\} \\
 & & \vdots \\
 & = & fix(\Gamma)\{i \mapsto 0\} \\
 & = & cond(B[i \neq 0]\{i \mapsto 0\}, \ldots, \lfloor\{i \mapsto 0\}\rfloor) \\
 & = & \lfloor\{i \mapsto 0\}\rfloor
\end{array}$$

[15] $D[s_1]\sigma \in \Sigma_\perp$

[9] $F : C \to D_\perp$
$g : D \to E$
$g \circ_\perp f = \lambda c.\textbf{let } x = f\,c.g\,x$
where $x \in D$ and $f \in D_\perp$

4.5 Extensions of WHILE

4.5.1 Local vars and procedures

Only static scoping!

$$\begin{array}{rcl} \text{Syntax} & : & \{\mathbf{var}\ x := a;\ s\} \\ & & \{\mathbf{proc}\ p = s_1;\ s_2\} \\ & & \mathbf{call}\ p \end{array}$$

Semantic domains:

$$\begin{array}{rcl} Venv & = & Var \overset{fin}{\rightarrow} Loc\ (\text{ve}) \\ Store & = & Loc \overset{fin}{\rightarrow} \mathbb{Z}\ (\sigma) \\ & & \sigma \circ ve \in \Sigma = Var \rightarrow \mathbb{Z} \end{array}$$

Procedure environment records semantics, not syntax!

$$Penv = Pname \overset{fin}{\rightarrow} (Store \rightarrow Store_\perp)$$

$$\begin{array}{rcl} D : Stm \rightarrow Venv \rightarrow Penv & \rightarrow & (Store \rightarrow Store_\perp) \\ D[\ \mathbf{skip}\ ]\ ve\ pe & = & \lambda\sigma.\lfloor\sigma\rfloor \\ D[x := a]\ ve\ pe & = & \lambda\sigma.\lfloor\sigma[ve(x) \mapsto A[a](\sigma \circ ve)]\rfloor \\ D[s_1;\ s_2]\ ve\ pe & = & D[s_2]\ ve\ pe \circ_\perp D[s_1]\ ve\ pe \\ D[\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\ ve\ pe & = & \lambda\sigma.cond(B[b](\sigma \circ ve), D[s_1]\ ve\ pe,\ \ .\overset{s_2}{.}.) \\ D[w]\ ve\ pe & = & \lambda\sigma.cond(B[b](\sigma \circ ve), (D[w]\ ve\ pe \circ_\perp D[s]\ ve\ pe)\sigma, \lfloor\sigma\rfloor) \\ D[\mathbf{call}\ p]\ ve\ pe & = & pe(p)\quad \text{(context condition: } pe(p) \text{ is defined,} \\ & & \qquad \text{i.e. all called procedures are defined in} \\ & & \qquad \text{the program text)} \\ D[\{\mathbf{var}\ x = a;\ s\}]\ ve\ pe & = & \lambda\sigma.D[s](ve[x \mapsto l])\ pe\ (\sigma[l \mapsto A[a](\sigma \circ ve)]) \\ & & \text{where } l \text{ is new, i.e. } l \notin dom(\sigma) \\ D[\{\mathbf{proc}\ p = s_1; s_2\}]\ ve\ pe & = & D[s_2]\ ve\ (pe[p \mapsto \Phi]) \\ & & \text{where } \Phi = D[s_1]\ ve\ (pe[p \mapsto \Phi])\quad \text{(least solution!)} \end{array}$$

Example:

12/19/2001

$$s = \{\mathbf{proc}\ p = \underbrace{(\mathbf{if}\ i = 0\ \mathbf{then}\ \mathbf{skip}\ \mathbf{else}\ (i := i - 1;\ \mathbf{call}\ p))}_{s_p};$$
$$\mathbf{call}\ p\}$$

$$\begin{array}{rcl} D[s]\{i \mapsto l\}\ \{\}\ \{l \mapsto 1\} & = & D[\mathbf{call}\ p]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \{l \mapsto 1\} \text{ where } \Phi = D[s_p]\{i \mapsto l\}\ \{p \mapsto \Phi\} \\ & = & \Phi\{l \mapsto 1\} = D[s_p]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \{l \mapsto 1\} \\ & = & D[i := i - 1;\ \mathbf{call}\ p]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \{l \mapsto 1\} \\ & = & \mathbf{let}\ \sigma' = D[i := i - 1]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \{l \mapsto 1\}.D[\mathbf{call}\ p]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \sigma' \\ & = & D[\mathbf{call}\ p]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \{l \mapsto 0\} \\ & = & \Phi\{l \mapsto 0\} = D[s_p]\{i \mapsto l\}\ \{p \mapsto \Phi\}\ \{l \mapsto 0\} \\ & = & \lfloor\{l \mapsto 0\}\rfloor \in Store_\perp \end{array}$$

4.5.2 Continuations and Exceptions

Continuations = (meaning of the) continuing computation = functional gotos
New statements for **WHILE** :

$$\textbf{escape}$$
$$s_1 \textbf{ handle } s_2$$

Example: (**if** $x = 0$ **then escape else** $x := x/x$) **handle** $x := 1$

Problem: $D[\textbf{escape}\,] = ?? \in \Sigma \to \Sigma_\perp$
Solution: continuations for normal and exceptional behavior.

Definition:
$Cont = \Sigma \to \Sigma_\perp$ = meaning of continuous computation
$D : Stm \to \underbrace{Cont}_{normal} \to \underbrace{Cont}_{exceptional} \to Cont$

Informal: $D[s]\ c\ d$ = execute s and continue $\begin{cases} \text{normally with c} \\ \text{exceptionally with d} \end{cases}$

$$\begin{aligned}
D[\,\textbf{skip}\,]\ c\ d &= c = \lambda\sigma.c\ \sigma \\
D[\,\textbf{escape}\,]\ c\ d &= d \\
D[x := a]\ c\ d &= \lambda\sigma.c(\sigma[x := A[a]\sigma]) \\
D[s_1;\ s_2]\ c\ d &= D[s_1](D[s_2]\ c\ d)\ d \\
D[s_1 \textbf{ handle } s_2]\ c\ d &= D[s_1]\ c\ (D[s_2]\ c\ d) \\
D[\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\ c\ d &= \lambda\sigma.cond(B[b]\sigma, D[s_1]\ c\ d\ \sigma, \ldots s_2 \ldots) \\
D[w]\ c\ d &= \lambda\sigma.cond(B[b]\sigma, D[s](D[w]\ c\ d)\ d\ \sigma, c\ \sigma)
\end{aligned}$$

Example:

$$\begin{aligned}
& D[(x := 1; \textbf{escape}\ ; x := 2) \textbf{ handle } x := 3]\ c\ d\ \sigma \\
= \ & D[(x := 1; \textbf{escape}\ ; x := 2) \textbf{ handle } x := 3]\ c\ \underbrace{(D[x := 3]\ c\ d)}_{d'}\ \sigma \\
= \ & D[x := 1](D[\textbf{escape}\ ; \ldots]\ c\ d')\ d'\ \sigma \\
= \ & D[\textbf{escape}\ ; \ldots]\ c\ d')\ d'\ (\sigma[x \mapsto 1]) \\
= \ & D[\textbf{escape}\,]\ (\ldots)\ d'\ (\sigma[x \mapsto 1]) = d'\ (\sigma[x \mapsto 1]) \\
= \ & D[x := 3]\ c\ d\ (\sigma[x \mapsto 1]) \\
= \ & c\ (\sigma[x \mapsto 3])
\end{aligned}$$

Top level call of D :
$D[s]\ (\lambda\sigma.\lfloor\sigma\rfloor)\ (\lambda\sigma.\perp)$ or
$D[s]\ (\lambda\sigma.\lfloor\sigma\rfloor)\ (\lambda\sigma.\lfloor\sigma[\underbrace{excn}_{Var} \mapsto 1]\rfloor)$

4.5.3 The Knaster-Tarski Fixpoint theorem

cpo $\supseteq$ complete lattice
continuity $\subseteq$ monotony

Definition:
Let $(P, \sqsubseteq)$ be a partial order, $X \subseteq P$.

- $p \in P$ is a *lower bound* for X iff. $\forall x \in X . p \sqsubseteq x$.
- $p \in P$ is a *greatest lower bound* for X iff. p is a lower bound and $q \sqsubseteq p$ for all lower bounds q of X.

(infimum = greatest lower bound)

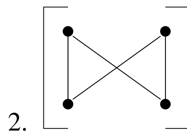
$\bigsqcap X$ = infimum of X, if it exists.

Definition: A partial order is a *complete lattice*[16] iff. every subset of P has an infimum.

Example:
$\wp A$ is a complete lattice with $\bigsqcap = \bigcap$.

$\bigcap \emptyset = A$ because any set $B \subseteq A$ is a lower bound for $\emptyset$, and thus A is the greatest lower bound.

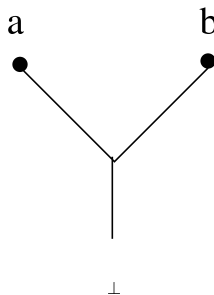
Counter-Example

1. [• •]
2. 
   Be careful! $P(A) : \bigsqcap \emptyset = A$!!! Because any set $B \subseteq A$ is a lower bound for $\emptyset$.

Remarks: If all infima exist, then all suprema exist as well:

$$\bigsqcup X = \bigsqcap \{u \in P \mid u \text{ is upper bound for } X\}$$

$$\begin{aligned}\bigcup \big\{\{1,2\},\{3,4\}\big\} &= \bigcap \big\{U \in \wp(\{1,\ldots,5\}) \mid U \supseteq \{1,2\},\{3,4\}\big\} \\ &= \bigcap \big\{\{1,2,3,4\},\{1,2,3,4,5\}\big\} = \{1,2,3,4\}\end{aligned}$$



$\bigsqcap \emptyset$ is the greatest element

[16] "Vollständiger Verband"

$\bigsqcup\emptyset$ is the least element

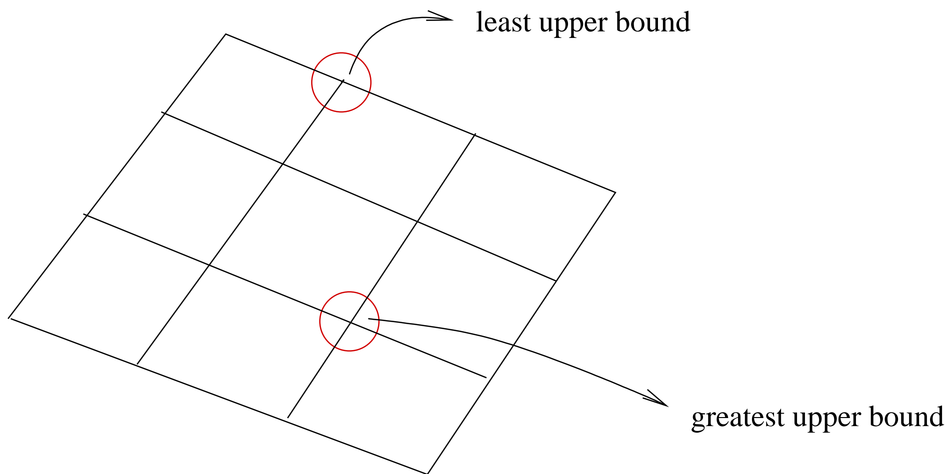
Trick Question:

- is $(\mathbb{N},\leq)$ a complete lattice???
- infimum: take least element in the subset
- supremum of $\mathbb{N}$???

Answer:

- $\bigsqcap\emptyset$ does not exist
- $\bigsqcup\mathbb{N}$ does not exist

$\Rightarrow$ not a complete lattice



Theorem:
Let $(L,\sqsubseteq)$ be a complete lattice, $f : L \to L$ be monotone.
Then f has a least fixpoint $m := \bigsqcap\{x \in L \mid f(x) \sqsubseteq x\}$ $\bigsqcap$ greatest lower bound

(Then $\{x \in L \mid f(x) = x\}$ is again a complete lattice.)

Proof: $f(m) = m, X := \{x \in L \mid f(x) \sqsubseteq x\}$

01/07/2002

1. $f(m) \sqsubseteq m$

   Let $x \in X \Rightarrow m \sqsubseteq x \overset{f \text{ monotone}}{\Rightarrow} f(m) \sqsubseteq f(x) \sqsubseteq x$
   $\Rightarrow f(m)$ is lower bound of X
   $\overset{m \text{ greatest lower bound}}{\Rightarrow} f(m) \sqsubseteq m$

2. $m \sqsubseteq f(m)$

   $f(m) \sqsubseteq m \overset{f \text{ monotone}}{\Rightarrow} f(f(m)) \sqsubseteq f(m)$
   $\Rightarrow f(m) \in X$
   $\overset{m=\bigsqcap X}{\Rightarrow} m \sqsubseteq f(m)$

3. m is *least* fixpoint:

   Let $f(p) = p$ be another fixpoint of f. Then $m \sqsubseteq p$:
   $f(p) = p \Rightarrow f(p) \sqsubseteq p \overset{\text{property of } X}{\Rightarrow} p \in X$
   $\overset{glb}{\Rightarrow} m \sqsubseteq p$

4.5.4 Nondeterminism

Extension of **WHILE** with nondeterministic choice:

$$s_1 \textbf{ or } s_2$$

Semantics: $S[s_1 \textbf{ or } s_2] = S[s_1] \cup S[s_2]$
where $S : Stm \rightarrow 2^{\Sigma \times \Sigma}$ is the relational semantics from section 4. There $S : Stm \rightarrow (\Sigma \nrightarrow \Sigma)$, but $\Sigma \nrightarrow \Sigma \subseteq 2^{\Sigma \times \Sigma}$ (Alternative: $Stm \rightarrow \Sigma \rightarrow 2^{\Sigma}$)
So what's new?
Example: $w_0 = \textbf{while } x \neq 0 \textbf{ do } (x := 0 \textbf{ or skip })$

$$\begin{aligned} S[w_0] &= \{(\sigma, \sigma[x \mapsto 0]) \mid \sigma \in \Sigma\} \\ S[x := 0] &= \{(\sigma, \sigma[x \mapsto 0]) \mid \sigma \in \Sigma\} \end{aligned}$$

w_0 may not terminate if **or** is "unfair".

Problem: S does not distinguish between *possible* termination (w_0) and *guaranteed* termination ($x := 0$)
Solution: $T : Stm \rightarrow 2^{\Sigma}$
$T[s]$ = set of start states that guarantee termination of s

$$\begin{aligned} T[\textbf{skip}] &= \Sigma \\ T[x := a] &= \Sigma \\ T[s_1 \textbf{ or } s_2] &= T[s_1] \cap T[s_2] \\ T[s_1; s_2] &= \{\sigma \in T[s_1] \mid \forall \sigma' (\sigma, \sigma') \in S[s_1] \Rightarrow \sigma' \in T[s_2]\} \\ &= \{\sigma \in T[s_1] \mid S[s_1]\sigma \subseteq T[s_2]\}^{17} \\ T[\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2] &= \{\sigma \mid \mathcal{B}[b]\sigma = \text{tt} \Rightarrow \sigma \in T[s_1] \wedge \mathcal{B}[b]\sigma = \text{ff} \Rightarrow \sigma \in T[s_2]\} \\ T[\underbrace{\textbf{while } b \textbf{ do } s}_{w}] &= T[\textbf{if } b \textbf{ then } s; w \textbf{ else skip }] \\ &= \{\sigma \mid \mathcal{B}[b]\sigma = \text{tt} \Rightarrow (\sigma \in T[s] \wedge S[s]\sigma \subseteq T[w]\} \quad \text{least solution} \\ &= fix(\Delta) \end{aligned}$$

where $\Delta : 2^{\Sigma} \rightarrow 2^{\Sigma}$
$\Delta(\mu) = \{\sigma \mid \mathcal{B}[b]\sigma = \text{tt} \Rightarrow (\sigma \in T[s] \wedge S[s]\sigma \subseteq \mu\}$
Does Δ have a least fixpoint? Yes!
Apply the KT-FP-theorem because $(2^{\Sigma}, \subseteq)$ is a complete lattice and Δ is monotone:
$\mu_1 \subseteq \mu_2 \Rightarrow \Delta(\mu_1) \subseteq \Delta(\mu_2)$

In some cases Δ is even continuous.

---

[17] Convention: If $R \subseteq A \times B, a \in A, X \subseteq A$: $R(a) = \{b \mid (a,b) \in R\}, R(X) = \{b \mid \exists a \in X.(a,b) \in R\} = \bigcup_{\text{ßn}X} R(a)$

Lemma: If $S[s]\sigma$ is finite ("finite nondeterminism") for all σ, then Δ is continuous.

Proof: Exercises!

In fact: Δ always continuous, but

- difficult to prove
- not true for other nondeterministic constructs, e.g. $random(x)$

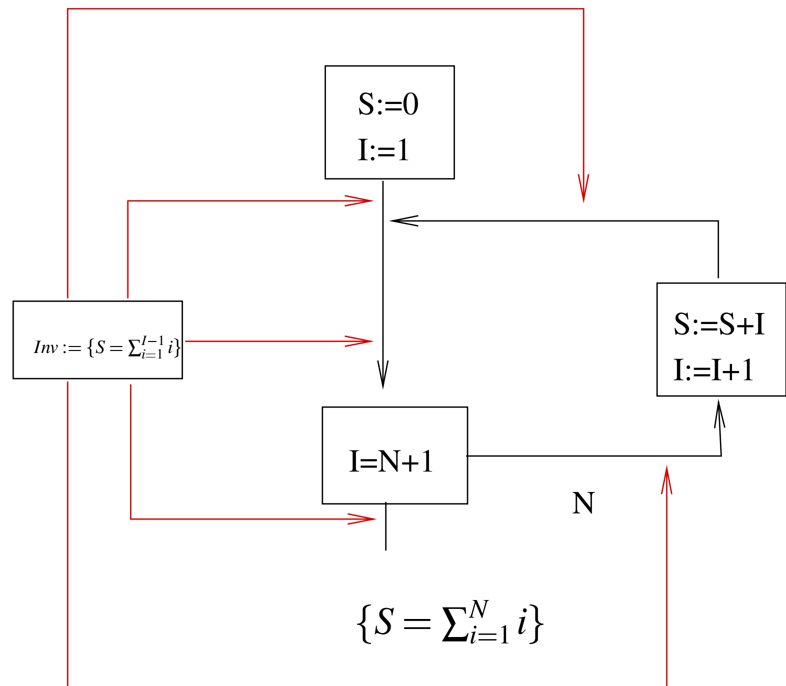
Example: $w_0 = \textbf{while } x \neq 0 \textbf{ do } (x := 0 \textbf{ or skip })$
$T[w_0] = ?$
Compute $fix(\Delta)$ by iteration: $\emptyset, \Delta(\emptyset), \Delta^2(\emptyset), \ldots$
Maybe find fixpoint in finite time / number of steps

$$\begin{aligned}
\Delta(\mu) &= \{\sigma \mid \sigma(x) \neq 0 \Rightarrow \sigma \in T[\ldots \textbf{ or } \ldots] \wedge S[\ldots \textbf{ or } \ldots]\sigma \subseteq \mu\} \\
&= \{\sigma \mid \sigma(x) \neq 0 \Rightarrow \{\sigma, \sigma[x \mapsto 0]\} \subseteq \mu\} \\
\Delta(\emptyset) &= \{\sigma \mid \sigma(x) = 0\} \\
\Delta^2(\emptyset) &= \Delta(\Delta(\emptyset)) \\
&= \Big\{\sigma \mid \sigma(x) \neq 0 \Rightarrow \underbrace{\{\sigma, \sigma[x \mapsto 0]\} \subseteq \{\sigma \mid \sigma(x) = 0\}}_{false}\Big\} \\
&= \{\sigma \mid \sigma(x) = 0\} = \Delta(\emptyset) = fix(\Delta)
\end{aligned}$$

Denotational Semantics for parallelism: tricky!

5 Axiomatic Semantics (Hoare-Logic; 1969)

01/09/2002



Invariant: $Inv = \{S = \sum_{i=1}^{I-1} i\}$

If we show that Inv holds at the last point (N), then the result is correct. To prove: If Inv is true before B, then Inv is true after B.

Assume Inv. To show: $S' = \sum_{i=1}^{I'-1} i$ where S', I' are the values of S, I after B.

$$\begin{aligned} S' &= S + I \\ &\overset{Inv}{=} \sum_{i=1}^{I-1} i \\ &= \sum_{i=1}^{I} i \\ &= \sum_{i=1}^{I'-1} i \text{ qed} \end{aligned}$$

Hoare logic:

$$S := 0; I := 1$$
$$\{S = \sum_{i=1}^{I-1} i\}$$
$$\textbf{while } I \neq N+1 \textbf{ do } (S := S+I; I := I+1)$$
$$\{S = \sum_{i=1}^{N} i\}$$

Hoare triple

$$\{\text{precondition}\} \text{ statement } \{\text{postcondition}\}$$

Intuitive meaning of $\{P\}S\{Q\}$:
if P is true in state σ, and execution of s takes σ to σ', then Q is true in σ'.

Partial correctness of $\{P\}S\{Q\}$

$$\{x = 0\}\textbf{while true do } x := x+1\{x = 0\}$$

is partially correct. Total correctness = partial correctness and termination

Examples of correct Hoare triples:

$$\begin{array}{ccl} \{I = 5\} & I := I+5 & \{I = 10\} \\ \{\textbf{ true }\} & I := 10 & \{I = 10\} \\ \{x = y\} & x := x+1 & \{x \neq y\} \text{ if } x \text{ and } y \text{ are distinct variables} \end{array}$$

Boundary cases

$\{$ **true** $\}$ s $\{$ **true** $\}$
partially correct for every s, total correct for any terminating s

$\{$ **true** $\}$ s $\{$ **false** $\}$
partially correct for nonterminating s, total correct for no s

$\{$ **false** $\}$ s $\{P\}$
partially correct for every s, total correct for any terminating s

The rules of Hoare logic (partial correctness):

$$\{P\} \quad \textbf{skip} \quad \{P\}$$
$$\underbrace{\{P[a/x]\}}_{P \text{ with } x \text{ replaced by } a} \quad x := a \quad \{P\}$$

$$\{x+2 = 7\}x := x+2\{x = 7\}$$

$$\frac{\{P\}s_1\{Q\} \qquad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

$$\frac{\{P \wedge b\}s_1\{Q\} \qquad \{P \wedge \neg b\}s_2\{Q\}}{\{P\}\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \{Q\}}$$

$$\frac{\{P \wedge b\} s \{P\} \qquad \text{"}P \text{ is invariant"}}{\{P\} w \{P \wedge \neg b\}}$$

$$\frac{P \Rightarrow P' \qquad \{P'\} s \{Q'\} \qquad Q' \Rightarrow Q}{\{P\} s \{Q\}}$$

strengthening precondition, weakening postcondition

$$\frac{\{x = 0\} s \{x = 5\}}{\{\ \textbf{false}\ \} s \{x < 10\}}$$

Derived rules

$$\frac{P \Rightarrow Q[a/x]}{\{P\} x := a \{Q\}}$$

$$\frac{x = 5 \Rightarrow (x+2) = 7}{\{x = 5\} x := x+2 \{x = 7\}}$$

$$\frac{P \Rightarrow Q[a/x] \qquad \{Q[a/x]\} x := a \{Q\} \qquad Q \Rightarrow Q}{\{P\} x := a \{Q\}}$$

$$\frac{P \Rightarrow Inv \qquad \{Inv \wedge b\} s \{Inv\} \qquad Inv \wedge \neg b \Rightarrow Q}{\{P\} w \{Q\}}$$

$$
\text{; rule}\ \dfrac{
\dfrac{\checkmark}{\{\ \mathbf{true}\ \}S := 0; I := 1\{S = \sum_{i=1}^{I-1} i\}}
\qquad
\dfrac{
S = \sum_{i=1}^{I-1} i \Rightarrow S = \sum_{i=1}^{I-1} i
\qquad
\dfrac{
\dfrac{S = \sum_{i=1}^{I-1} i \wedge I \neq N+1 \Rightarrow S+I = \sum_{i=1}^{I} i}{\{S = \sum_{i=1}^{I-1} i \wedge I \neq N+1\}S := S+I\{S = \sum_{i=1}^{I} i\}}
\qquad
\{S = \sum_{i=1}^{I} i\}I := I+1\{Inv\}
}{\{S = \sum_{i=1}^{I-1} i \wedge I \neq N+1\}S := S+I; I := I+1\{S = \sum_{i=1}^{I-1} i\}}
\qquad
S = \sum_{i=1}^{I-1} i \wedge \neg(I \neq N+1) \Rightarrow S = \sum_{i=1}^{N} i
}{\{S = \sum_{i=1}^{I-1} i\}\mathbf{while}\ \ldots\ \{S = \sum_{i=1}^{N} i\}}
}{\{\ \mathbf{true}\ \}S := 0; I := 1; \mathbf{while}\ I \neq N+1\ \mathbf{do}\ (S := S+I; I := I+1)\{S = \sum_{i=1}^{N} i\}}
$$

5.1 Assertions

"Zusicherungen"

The *syntactic* (intensional) approach [WINSKEL93]:

- assertions = syntactic formulae from some logical system (usually: FOL[18] + $\mathbb{Z}$ = first order arithmetic)

The *semantic* (extensional) approach [NIELSON99]:

- assertions = predicates on $\Sigma = \Sigma \to T \quad (T = \{\text{tt}, \text{ff}\})$

Notation: abbreviate $(P = \text{tt})$ by P.

Example:

$$\begin{aligned} \text{syntactic} &: \; x > y \\ \text{semantic} &: \; \lambda\sigma.\sigma(x) > \sigma(y) : \Sigma \to T \end{aligned}$$

Note: $\Sigma \to T$ is the set of *all* functions, not just those expressible in some fixed language (e.g. *Bexp*)

5.2 The semantic approach

Definition: (Validity/Truth)

$$\begin{aligned} \models \{P\}s\{Q\} \quad &\Leftrightarrow \quad \forall \sigma, \sigma'. P(\sigma) \wedge \langle s, \sigma \rangle \to \sigma' \Rightarrow Q(\sigma') \\ &\Leftrightarrow \quad \forall \sigma, \sigma'. \langle s, \sigma \rangle \to \sigma' \Rightarrow (P\,\sigma \Rightarrow Q\sigma') \end{aligned}$$

"$\{P\}s\{Q\}$ is valid"

$\vdash \{P\}s\{Q\}$ means the triple $\{P\}s\{Q\}$ is *provable/derivable* in the following system:

$$\vdash \{P\}\textbf{skip}\ \{P\}$$

;-rule unchanged

$$\vdash \frac{\{\lambda\sigma.P\sigma \wedge \mathcal{B}[b]\sigma\}s_1\{Q\}}{\{P\}\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\{Q\}}$$

$$\vdash \frac{\{\lambda\sigma.P\ \sigma \wedge \mathcal{B}[b]\sigma\}s\{P\}}{\{P\}w\{\lambda\sigma.P\ \sigma \wedge \neg\mathcal{B}[b]\sigma\}}$$

before: $\vdash \{P[a/x]\}x := a\{P\}$

now: $\vdash \{\underbrace{\lambda\sigma.P(\sigma[x \mapsto \mathcal{A}[a]\sigma])}_{P[x \mapsto a]}\}x := a\{P\}$

$$\frac{\forall\sigma.P\sigma \Rightarrow P'\sigma \qquad \vdash \{P'\}s\{Q'\} \qquad \forall\sigma.Q'\sigma \Rightarrow Q\sigma}{\vdash \{P\}s\{Q\}}$$

Wanted: soundness: $\vdash \{P\}s\{Q\} \Rightarrow \;\models \{P\}s\{Q\}$
completeness: $\vdash \{P\}s\{Q\} \Leftarrow \;\models \{P\}s\{Q\}$

01/14/2002

5.3 Soundness

Theorem: $\vdash \{P\}s\{Q\} \Rightarrow \ \models \{P\}s\{Q\}$

Proof: by rule induction on $\vdash \{P\}s\{Q\}$,
i.e. by showing that each rule preserves validity ($\models$)
Two cases:

- **:=** To show $\models \{P[x \mapsto a]\}x := a\{P\}$

$$\begin{aligned} &\Leftrightarrow \quad \forall \sigma, \sigma'. \langle x := a, \sigma\rangle \to \sigma' \Rightarrow (P(\sigma[x \mapsto \mathcal{A}[a]\sigma]) \Rightarrow P\sigma') \\ &\Leftrightarrow \quad \forall \sigma, \sigma'. \sigma' = \sigma[x \mapsto \mathcal{A}[a]\sigma] \Rightarrow (P(\sigma[x \mapsto \mathcal{A}[a]\sigma]) \Rightarrow P\sigma') \quad \checkmark \end{aligned}$$

- **while** :

  (A) Assume $\models \{\lambda\sigma.P\sigma \wedge \mathcal{B}[b]\sigma\}s\{P\}$

  (B) To show: $\models \{P\}w\{\lambda\sigma.P\sigma \wedge \neg\mathcal{B}[b]\sigma\}$

  Lemma: If $\langle w, \sigma\rangle \to \sigma', P\sigma, \forall \sigma, \sigma'. P\sigma \wedge \mathcal{B}[b]\sigma \wedge \langle s, \sigma\rangle \to \sigma' \Rightarrow P\sigma'$ then $P\sigma' \wedge \neg\mathcal{B}[b]\sigma'$

  Proof: by induction on $\langle w, \sigma\rangle \to \sigma'$
  Proof of (B): Assume $\langle w, \sigma\rangle \to \sigma', P\sigma$. Then by lemma
  and (A) we have $P\sigma' \wedge \neg\mathcal{B}[b]\sigma'$ qed.

5.4 Completeness

Theorem: $\models \{P\}s\{Q\} \Rightarrow \vdash \{P\}s\{Q\}$
Proof idea based on weakest preconditions.

Definition:

$$\begin{aligned} wp\ s\ Q &= \lambda\sigma.\forall\sigma'.\langle s, \sigma\rangle \to \sigma' \Rightarrow Q\ \sigma' \\ &= \text{the set of all } \sigma \text{ that lead into } Q \text{ when executing } s \end{aligned}$$

(often: weakest *liberal* precondition)

Fact:

- $\models \{P\}s\{Q\}$ iff. $\forall\sigma P\sigma \Rightarrow wp\ s\ Q\sigma$

Lemmas:

- wp **skip** $Q = Q$
- $wp\ (x := a)\ Q = Q[x \mapsto a]$
- $wp\ (s_1; s_2)\ Q = wp\ s_1\ (wp\ s_2\ Q)$

- $wp\ (\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2)Q = (\lambda\sigma.(\mathcal{B}[b]\sigma \Rightarrow wp\ s_1\ Q\sigma) \wedge (\neg\mathcal{B}[b]\sigma \Rightarrow wp\ s_2\ Q\sigma))$
- $wp\ (w)\ Q\sigma = \begin{cases} Q\sigma & \text{if } \neg\mathcal{B}[b]\sigma \\ wp\ (s;w)\ Q\sigma = wp\ s\ (wp\ w\ Q)\sigma & \text{if } \mathcal{B}[b]\sigma \end{cases}$

$$\text{consequence rule}\ \frac{\dfrac{\models \{P\}s\{Q\}}{\forall\sigma.P\sigma \Rightarrow wp\ s\ Q\sigma} \qquad \dfrac{\text{Lemma}}{\vdash \{wp\ s\ Q\}s\{Q\}} \qquad \overset{\checkmark}{\forall\sigma.\ Q\sigma \Rightarrow Q\sigma}}{\vdash \{P\}s\{Q\}}$$

Lemma: $\forall Q. \vdash \{wp\ s\ Q\}\ s\ \{Q\}$

Proof: by induction on s

1. $\vdash \{wp\ \textbf{skip}\ Q\}\ \textbf{skip}\ \{Q\}$ ✓
2. $\vdash \{wp\ (x := a)\ Q\}\ x := a\ \{Q\}$ ✓
3. $$\frac{\overset{IH}{\vdash \{wp\ s_1\ (wp\ s_2\ Q)\}\ s_1\ \{wp\ s_2\ Q\}} \qquad \overset{IH}{\vdash \{wp\ s_2\ Q\}\ s_2\ \{Q\}}}{\{wp\ (s_1;s_2)\ Q\}\ s_1;s_2\ \{Q\}}$$
4. $$\frac{\vdash \{\lambda\sigma.\overbrace{\mathcal{B}[b]\sigma \wedge wp\ (\textbf{if}\ \dots)\ \sigma}^{\mathcal{B}[b]\sigma\wedge wp\ s_1\ Q\sigma}\}\ s_1\ \{Q\}}{\{wp\ (\textbf{if}\ \dots)\ Q\}\ \textbf{if}\ \dots\ \{Q\}}$$

$$IH_1:\ \text{conseq.}\ \frac{\vdash \{wp\ s_1\ Q\}\ s_1\ \{Q\}}{\vdash \{\lambda\sigma.\mathcal{B}[b]\sigma \wedge wp\ s_1\ Q\sigma\}\ s_1\ \{Q\}}$$

5. $$\frac{\dfrac{\text{conseq.}\ \dfrac{\overset{IH}{\vdash \{wp\ s\ (wp\ w\ Q)\}\ s\ \{wp\ w\ Q\}}}{\vdash \{wp\ w\ Q \wedge \mathcal{B}[b]\sigma\}\ s\ \{wp\ w\ Q\}}}{\vdash \{wp\ w\ Q\}\ w\ \{\lambda\sigma.wp\ w\ Q\sigma \wedge \neg\mathcal{B}[b]\sigma\}}}{\vdash \{wp\ w\ Q\}\ w\ \{Q\}}$$

$$\begin{aligned} & wp\ w\ Q\sigma \wedge \neg\mathcal{B}[b]\sigma \\ \overset{Lemma}{=}\ & Q\sigma \wedge \neg\mathcal{B}[b]\sigma \Rightarrow Q\sigma \end{aligned}$$

$$\begin{aligned} & wp\ w\ Q\sigma \wedge \mathcal{B}[b]\sigma \\ \overset{Lemma}{=}\ & wp\ s\ (wp\ w\ Q)\sigma \wedge \mathcal{B}[b]\sigma \\ \Rightarrow\ & wp\ s\ (wp\ w\ Q)\sigma \end{aligned}$$

5.5 Expressiveness

Problem: Proof of $\vdash \{P\}s\{Q\}$ with $P,Q \in A$ (A set of formulae) may require assertions not in A.

Contrived example: $A = \{\textbf{true}\} \cup \{X = n \mid X \in \textbf{var} \wedge n \in \mathbb{Z}\}$
(*Bexp* are restricted to A)

$$
\begin{array}{rcl}
\models \{\textbf{true}\} & & \\
 & & \textbf{if } y = 0 \textbf{ then } x := 0 \textbf{ else } x := 1 \\
\{x = 0 \vee x = 1\} & \leftarrow & \text{not in } A \\
 & & \textbf{if } x = 0 \textbf{ then skip} \\
 & & \textbf{else if } x = 1 \textbf{ then } x := 0 \textbf{ else } x := 2 \\
\{x = 0\} & & \\
 & & \Rightarrow \text{ not provable!}
\end{array}
$$

Another example: Presburger arithmetic = FOL + $(\mathbb{N}, +)$
Problems: programs can synthesize $*$ from $+$, but not FOL

$$
\begin{array}{rcl}
\{p = 0 \wedge x \geq 0 \wedge y \geq 0\} & & \\
 & & \textbf{while } x > 0 \textbf{ do } (p := p + y;\ x := x - 1) \\
\{p = x * y\} & \leftarrow & \text{not in Presburger arithmetic} \\
 & & \textbf{while } p > 0 \textbf{ do } p := p - y \\
\{p = 0\} & & \\
 & & \Rightarrow \text{ valid but not provable}
\end{array}
$$

Definition: $A \subseteq \Sigma \to T$ is *expressive* iff. for every statement s and $Q \in A$

01/16/2002

$$wp\ s\ Q \in A$$

Corollary: If

- A is expressive,
- A contains *Bexp* (for all $b \in Bexp$, $\lambda\sigma.\mathcal{B}[b]\sigma \in A$), and
- A is closed under $\wedge$ and $\neg$ ($P, Q \in A$ then $\lambda\sigma.P\sigma \wedge Q\sigma \in A$)

then $\models \{P\}s\{Q\}$ with $P, Q \in A$ implies there is a proof of $\vdash \{P\}s\{Q\}$ containing only assertions in A.

Theorem: First-order arithmetic (FOA, Peano Arithmetic), i.e. FOL + $(\mathbb{N}, +, *, <)$, is expressive.

Proof: (ideas)

Gödel

1. Program state can be encoded as a natural number.
2. Program execution can be encoded as a FOA formula. For all s there is a formula $P_s(\sigma, \sigma')$ in FOA such that $P_s(\sigma, \sigma') \iff \langle s, \sigma \rangle \to \sigma'$
(Beware: Definition of $\langle \ldots,\ \ldots \rangle \to \ldots$ is not in FOA, because leastness of the relation is not expressible in FOA.)

$$
\begin{array}{rcl}
w & = & \textbf{while } b \textbf{ do } s \\
P_w(\sigma, \sigma') & = & \exists \text{ a sequence of states } \overline{\sigma} \text{ of length n .} \\
 & & \overline{\sigma_0} = \sigma \wedge \overline{\sigma_n} = \sigma' \wedge \neg\mathcal{B}[b]\sigma' \wedge \forall i < n.\mathcal{B}[b]\sigma_i \wedge P_s(\overline{\sigma_i}, \overline{\sigma_{i+1}})
\end{array}
$$

Question:

- Presburger Arithmetic contains $+$ but not $*$.
- Programs can synthesize $*$ from $+$.

$\Rightarrow$ Presburger Arithmetic is not expressive.

- FOA contains $+,*$ but not exponentiation.
- Programs can synthesize exponentiation from $*$.
- **Why is FOA expressive?**

Answer: Because there is a formula $E(n,m,e)$ such that $E(n,m,e) \iff e = n^m$

5.6 Relative Completeness

Intuition: finite proof system $\approx$ Prolog program

$\Rightarrow$ Main property of a finite proof system is that all provable formulae are recursively enumerable[19] (r.e.)

Theorem: If $\vdash$ is sound and recursively enumerable then it is not complete.

Proof: Remember: $\models \{ \textbf{ true } \} s \{ \textbf{ false } \}$ iff. s never terminates.

Theorem: $\{s \mid s \text{ never terminates}\}$ is not recursively enumerable.

(**Lemma:** $\{s \mid s \text{ terminates somewhere}\}$ *is* recursively enumerable.)

Assume $\vdash$ is sound, complete and recursively enumerable

$$\begin{array}{rcl} \vdash \text{ r.e.} & \Rightarrow & \{s \mid \ \vdash \{\textbf{true }\} s \{\textbf{false }\}\} \text{ is r.e.} \\ \vdash sound\&compl. & \Rightarrow & \{s \mid \ \models \{\textbf{true }\} s \{\textbf{false }\}\} \text{ is r.e.} \quad ↯ \quad qed. \end{array}$$

Thus our proof system for Hoare Logic cannot be recursively enumerable
Problem: $P \Rightarrow P'$ and $Q' \Rightarrow Q$ in consequence rule refer to truth, but not to some proof system.

Theorem: (Gödel)
The true formulae of FOA are not recursively enumerable, i.e. there is no finite, sound and complete proof system for FOA.

Choice in consequence rule: $P \Rightarrow P'$ and $Q' \Rightarrow Q$ refer to

1. truth in FOA. Then $\vdash$ is not recursively enumerable
2. provability in some standard proof system for FOA. Then $\vdash$ is incomplete.

Thus Hoare Logic is *relatively* complete with respect to (in-)completeness of the assertion logic.

[19]"rekursiv aufzählbar"

5.7 Verification condition generation

Extended syntax: $w = \textbf{while}\ b\ \textbf{do}\ \{I\}s$
Function *pre* is very similar to *wp* (weakest precondition) with exception of the new syntax for the while statement

$$
\begin{array}{rcl}
pre : Stm \to Assn & \to & Assn \\
pre\ \textbf{skip}\ Q & = & Q \\
pre\ (x := a)\ Q & = & Q[a/x] \\
pre\ (s_1;s_2)\ Q & = & pre\ s_1\ (pre\ s_2\ Q) \\
pre\ (\textbf{if}\ \ldots)\ Q & = & (b \Rightarrow pre\ s_1\ Q) \wedge \\
 & & (\neg b \Rightarrow pre\ s_2\ Q) \\
pre\ w\ Q & = & I
\end{array}
$$

As is clear, not all annotated partial correctness assertions are valid. To be so it is sufficient to establish the validity of certain assertions, called *verification conditions* for which all mention of commands is eliminated.

from Winskel book

$$
\begin{array}{rcl}
vc : Stm \to Assn & \to & Assn \quad vc : \text{verification condition} \\
 & & (\text{Soundness: } vc\ s\ Q \Rightarrow\ \vdash \{pre\ s\ Q\}s\{Q\}) \\
vc\ \textbf{skip}\ Q & = & \textbf{true} \\
vc\ (x := A)\ Q & = & \textbf{true} \\
vc\ (s_1;s_2)\ Q & = & vc\ s_1\ (pre\ s_2\ Q) \wedge vc\ s_2\ Q \\
vc\ (\textbf{if}\ \ldots)\ Q & = & vc\ s_1\ Q \wedge vc\ s_2\ Q \\
vc\ w\ Q & = & \underbrace{(I \wedge b \Rightarrow pre\ s\ I)}_{\text{"}I\text{ is invariant"}} \wedge (I \wedge \neg b \Rightarrow Q) \wedge (vc\ s\ I)
\end{array}
$$

Example:

$$
\begin{array}{rcl}
w & = & \textbf{while}\ I \neq N+1\ \textbf{do}\ \{P\}\ s \\
s & = & S := S+I;\ I := I+1 \\
P & = & (S = \sum_{i=1}^{I-1} i) \\
Q & = & (S = \sum_{i=1}^{N} i) \\
pre\ w\ Q & = & P \quad (P \text{ true e.g. if } I = 1 \text{ and } S = 0) \\
vc\ w\ Q & = & (P \wedge I \neq N+1 \Rightarrow pre\ s\ P) \wedge \\
 & & (P \wedge I = N+1 \Rightarrow Q) \wedge \\
 & & vc\ s\ P \\
vc\ s\ P & = & \textbf{true}\ \wedge \textbf{true} \quad \checkmark \\
pre\ s\ P & = & pre\ (S := S+I)\ (pre\ (I := I+1)\ P) \\
 & = & (S = \sum_{i=1}^{I} i) \\
 & = & \underbrace{S + I = \sum_{i=1}^{I} i}_{=P}
\end{array}
$$

Theorem: $vc\ s\ B \Rightarrow \vdash \{pre\ s\ B\}s\{B\}$

01/21/2002

Proof: by induction on s.

- $s = \textbf{skip}$: $vc\ \textbf{skip}\ B = \textbf{true} \Rightarrow \vdash \{\overbrace{pre\ \textbf{skip}\ B}^{B}\}\textbf{skip}\ \{B\}$
- $s = s_1;s_2$: $vc\ (s_1;s_2)\ B = vc\ s_1\ (pre\ s_2\ B) \wedge vc\ s_2\ B$
  $\overset{IH}{\Rightarrow} \vdash \{\underbrace{pre\ s_1\ (pre\ s_2\ B)}_{=pre\ (s_1;s_2)\ B}\}s_1\{pre\ s_2\ B\}$ and $\vdash \{pre\ s_2\ B\}s_2\{B\}$
  $\overset{HL}{\Rightarrow} \vdash \{pre\ (s_1;s_2)\ B\}s_1;s_2\{B\}$
- $s = \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2$: $vc\ s\ B = vc\ s_1\ B \wedge vc\ s_2\ B$
  $\overset{IH}{\Rightarrow} \vdash \{pre\ s_i\ B\}s_i\{B\} \overset{conseq.}{\Rightarrow} \vdash \{b \wedge pre\ s_1\ B\}s_1\{B\} \wedge \vdash \{\neg b \wedge pre\ s_2\ B\}s_2\{B\}$
  precondition strengthening

  $$\left(\text{if-then-else-rule}\ \frac{\vdash \{b \wedge pre\ s_1\ B\}s_1\{B\} \qquad \vdash \{\neg b \wedge pre\ s_2\ B\}s_2\{B\}}{\vdash \{pre\ s\ B\}s\{B\}}\right)$$

- $s = \textbf{while}\ b\ \textbf{do}\ \{I\}s_0$
  $vc\ s\ B = (I \wedge \neg b \Rightarrow B) \wedge (I \wedge b \Rightarrow pre\ s_0\ I) \wedge vc\ s_0\ I$

$$\text{conseq}\ \frac{I \Rightarrow I \qquad \text{w-rule}\ \dfrac{\text{conseq}\ \dfrac{I \wedge b \Rightarrow pre\ s_0\ I \qquad \dfrac{vc\ s_0\ I \wedge I.H.}{\vdash \{pre\ s_0\ I\}s_0\{I\}} \qquad I \Rightarrow I}{\vdash \{I \wedge b\}s_0\{I\}}}{\vdash \{I \wedge \neg b\}w\{I \wedge \neg b\}} \qquad I \wedge \neg b \Rightarrow B}{\vdash \{\underbrace{pre\ w\ B}_{=I}\}w\{B\}}$$

Corollary: $\left\{ \begin{array}{l} vc\ s\ B \wedge \\ A \Rightarrow pre\ s\ B \end{array} \right\} \Rightarrow \vdash \{A\}s\{B\}$

! Only helpful if *pre* and *vc* "sensible".

- $pre\ s\ B = \textbf{false}$ not interesting ($\{\ \textbf{false}\ \}s\{Q\}$ always provable)
- $vc\ s\ B = \textbf{false}$ not interesting

Needed: completeness!

Theorem: If $\vdash \{A\}s\{B\}$ (s is not annotated), then there is an annotated version s' of s such that $vc\ s'\ B$ and $A \Rightarrow pre\ s'\ B$.

Proof: by rule induction on $\vdash \{A\}s\{B\}$

- Case w-rule:

  $$\frac{\dfrac{\vdots}{\vdash \{A \wedge b\}s\{A\}}}{\vdash \{A\}w\{\underbrace{A \wedge \neg b}_{B}\}} \qquad (w = \textbf{while}\ b\ \textbf{do}\ s)$$

  $IH \Rightarrow \exists$ an annotated version s' of s such that $vc\ s'\ A$ and $A \wedge b \Rightarrow pre\ s'\ A$

  $w' = \textbf{while}\ b\ \textbf{do}\ \{A\}s \quad (\Rightarrow (A \Rightarrow pre\ w'\ B))$
  $vc\ w'\ B = (A \wedge \neg b \Rightarrow B) \wedge (A \wedge b \Rightarrow pre\ s'\ A) \wedge vc\ s'\ A$

- Case conseq. rule:

  $$\frac{A \Rightarrow A' \quad \vdash \{A'\}s\{B'\} \quad B' \Rightarrow B}{\vdash \{A\}s\{B\}}$$

  IH : $\exists$ annotated version of s' of s such that $\underbrace{vc\ s'\ B'}_{\Rightarrow vc\ s'\ B}$ and $\underbrace{A' \Rightarrow pre\ s'\ B'}_{\Uparrow\ \Downarrow\ A \Rightarrow pre\ s'\ B}$ by lemma:

  Lemma: If $P \Rightarrow Q$ then $vc\ s\ P \Rightarrow vc\ s\ Q$ and $pre\ s\ p \Rightarrow pre\ s\ Q$.

5.8 Total Correctness

Total correctness = partial correctness $\wedge$ termination

$\models_t \{A\}s\{B\}$ iff. $\models \{A\}s\{B\}$ and

1. $\forall \sigma. A\sigma \Rightarrow \exists \sigma'. \langle s,\sigma\rangle \rightarrow \sigma'$
   only for deterministic languages, or
2. $(\langle s,\sigma\rangle \rightarrow_1 \langle s',\sigma'\rangle$: small-step semantics)
   there is no infinite chain $\langle s,\sigma\rangle \rightarrow_1 \langle s_1,\sigma_1\rangle \rightarrow_1 \langle s_2,\sigma_2\rangle \rightarrow_1 \ldots$ such that $A\sigma$ holds, or
3. $\forall \sigma. A\sigma \Rightarrow s \downarrow \sigma$ ("s started in σ *must* terminate")
   (analogous to $\sigma \in T[s]$)

   Inductive definition of $\downarrow$:

   - **skip** $\downarrow \sigma$
   - $(x := a) \downarrow \sigma$
   - $\dfrac{s_1 \downarrow \sigma \quad \forall \sigma' \langle s,\sigma\rangle \rightarrow \sigma' \Rightarrow s_2 \downarrow \sigma'}{s_1; s_2 \downarrow \sigma}$
   - $\dfrac{s_1 \downarrow \sigma \quad \mathcal{B}[b]\sigma}{\textbf{if}\ b\ \ldots \downarrow \sigma} \quad \dfrac{s_2 \downarrow \sigma \quad \neg\mathcal{B}[b]\sigma}{\textbf{if}\ b\ \ldots \downarrow \sigma}$
   - $\dfrac{\neg\mathcal{B}[b]\sigma}{w \downarrow \sigma} \quad \dfrac{\mathcal{B}[b]\sigma \quad (s;w) \downarrow \sigma}{w \downarrow \sigma}$

Proof rules for total correctness:

$$\frac{\{A \wedge b \wedge t = z\}s\{A \wedge t < z\} \quad A \Rightarrow t \geq 0}{\vdash \{A\}w\{A \wedge \neg b\}} \tag{8}$$

where $t \in Aexp$ and z is a "new" variable, i.e. z does not occur in A, b, t.

01/23/2002

This rule is *seriously incomplete* if $t \in Aexp$ because $Aexp$ can only express polynomials, t is an upper bound on the running time of the loops, but we can program loops with super-polynomial running time.

Alternative: extend set of functions allowed to occur in t.
Better: new proof rule without explicit t:

$$\frac{\vdash \{A(n+1)\}s\{A(n)\}}{\vdash \{\exists n \in \mathbb{N}. A(n)\}w\{A(0)\}} \tag{9}$$

if $A(0) \Rightarrow \neg b$ and $A(n+1) \Rightarrow b$ (side conditions).
This rule is sound and in a certain sense complete (for first order arithmetic).

Example: for (8)

$$\frac{\{A \land N = 0 \land N = Z\}\ N := N-1;\ M := M+1\ \{A \land N < Z\} \qquad A \Rightarrow N \geq 0}{\vdash \{\underbrace{N \geq 0 \land N + M = C}_{A}\}\ \textbf{while}\ N \neq 0\ \textbf{do}\ (N := N-1;\ M := M+1)\ \{A \land N = 0\}}$$

Example: for (9)

$$\text{conseq}\frac{\dfrac{\vdash \{x = n+1\} x := x-1 \{x = n\}}{\vdash \{\exists n. x = n\} w \{x = 0\}}}{\vdash \{x \geq 0\} \textbf{while}\ x \neq 0\ \textbf{do}\ x := x-1 \{x = 0\}}$$

side conditions: $x = 0 \Rightarrow \neg(x \neq 0)$, $x = n+1 \Rightarrow x \neq 0$.

Example:

$$\text{conseq}\frac{\dfrac{\dfrac{x + (n+1)*k \geq a > x + n*k \Rightarrow (x+k) + n*k \geq a > (x+k) + (n-1)*k}{\vdash \{x + (n+1)*k \geq a > x + n*k\}\ x := x+k\ \{x + n*k \geq a > x + (n-1)*k\}}}{\vdash \{\exists n. x + n*k \geq a > x + (n-1)*k\}\ w\ \{x + 0*k \geq a \geq x - k\}}}{\vdash \{\ \textbf{true}\ \}\ \textbf{while}\ x < a\ \textbf{do}\ x := x + \underbrace{k}_{\in \mathbb{N}}\ \{x \geq a\}}$$

side conditions: $x \geq a \Rightarrow \neg x < a$ and $x + (n+1)*k \geq a > x + n*k \Rightarrow x < a$

Remark: $S_1\{N = "f(M)"\}$ **while** $X < N +$ **do** $X := X + 1$
there is a predicate F such that $N = f(M)$ iff. $F(M, N)$.

$$\frac{}{\{\exists n. f(M) - X = n\}\ w\ \{X = N\}}$$

$\{\exists n. F(M, N) \land N - X = n\}$

5.9 Extension of WHILE

5.9.1 Nondeterminism

s_1 **or** s_2

$$\frac{\vdash \{P\}\ s_1\ \{Q\} \qquad \vdash \{P\}\ s_2\ \{Q\}}{\vdash \{P\}\ s_1\ \textbf{or}\ s_2\ \{Q\}}$$

```
while X != 0 do
  if X < 0 then X := some arbitrary positive number
           else X := X-1
```

terminates, but number of iterations is not a function of X, is unbounded.
Termination measure: $(X < 0, X)$
Ordering:

$$\begin{array}{rcl} (\textbf{true}, _) & > & (\textbf{false}, _) \\ (\textbf{false}, m) & > & (\textbf{false}, n) \text{ iff. } m > n \end{array}$$

Example product of two orderings.

5.9.2 Arrays

Extended syntax:

$$\begin{array}{rcl} Aexp & ::= & \ldots \mid A[a] \\ Stm & ::= & \ldots \mid A[a_1] := a_2 \end{array}$$

$A, A_1, A_2, \ldots$ are array variables.
$\mathcal{A}$ is the evaluation function.

Problem: assignment axiom not valid:

$$\begin{array}{lcl} \not\models \{0 = 0\} & A[A[1]] := 0 & \{A[A[1]] = 0\} \\ A[1] = 1 & & A[A[1]] = 42 \\ A[0] = 42 & & 0 \end{array}$$

Solution: regard arrays as *one* variable, not as a sequence of variables $A[1], A[2], \ldots$
$\Rightarrow$ Assignment must change A as a whole.

Model: array = $\mathbb{Z} \to \mathbb{Z}$
Translate **WHILE** + array $\mapsto$ **WHILE** + function variables

1. Replace $A[a_1] := a_2$ by $A := A[a_1 \mapsto a_2]$
2. replace $A[a]$ by $A(a)$

Result: **WHILE** program in an extended set of $Aexp$:

$$\begin{array}{rcl} Aexp & ::= & \ldots \mid f(a) \\ Funs & ::= & A_1 \mid A_2 \mid \ldots \mid f[a_1 \mapsto a_2] \quad f \in Funs \end{array}$$

Example: $A[A[1]] := 0$

$$\begin{array}{ll} \mapsto & A := A[A[1] \mapsto 0] \\ \mapsto & A := A[A(1) \mapsto 0] \\ & NB : A[_ \mapsto _] \text{ part of programming language syntax!} \end{array}$$

To prove $\vdash \{A[1] \neq 1\}\ A[A[1]] := 0\ \{A[A[1]] = 0\}$
we prove $\vdash \{A(1) \neq 1\}\ A := A[A(1) \mapsto 0]\ \{A(A(1)) = 0\}$
Need to show: $A(1) \neq 1 \Rightarrow A'(A'(1)) = 0$
Assume $A(1) \neq 1$

$$\begin{array}{rcl} A'(A'(1)) & \overset{Def.of A' \wedge A(1) \neq 1}{=} & A'(A(1)) \\ & \overset{Def.of A'}{=} & 0 \end{array}$$

01/28/2002

5.9.3 Procedures

1 procedure without parameters
New statement: **call**
Implicit procedure definition: **proc** = body
Operational semantics:

$$\frac{\langle body, \sigma\rangle \to \sigma'}{\langle \mathbf{call}\,, \sigma\rangle \to \sigma'}$$

New format of Hoare rules:

$$H \vdash \{P\}\ s\ \{Q\} \tag{10}$$

where H[20] is 0 or 1 Hoare triple (of the form $\{A\}$ **call** $\{B\}$).
(10) "$\{P\}\ s\ \{Q\}$ is provable under the assumption H"

Adaption of old proof rules to a new format: add $H \vdash$ everywhere:

$$\frac{P \Rightarrow P' \qquad H \vdash \{P'\}\ s\ \{Q'\} \qquad Q' \Rightarrow Q}{H \vdash \{P\}\ s\ \{Q\}}$$

New rule for **call**

$$\frac{\overbrace{\{P\}\ \mathbf{call}\ \{Q\}}^{P(n)} \vdash \overbrace{\{P\}\ body\ \{Q\}}^{P(n+1)}}{\vdash \{P\}\ \mathbf{call}\ \{Q\}}$$

Assumption rule: $\{P\}\ s\ \{Q\} \vdash \{P\}\ s\ \{Q\}$

Example: (1)

$$\text{call rule}\ \frac{\text{assumption rule}\ \dfrac{body = \mathbf{call}}{\{P\}\ \mathbf{call}\ \{Q\} \vdash \{P\}\ \mathbf{call}\ \{Q\}}}{\vdash \{P\}\ \mathbf{call}\ \{Q\}}$$

Example:

$$\begin{array}{llll} body = \mathbf{if}\ I = 0 & \mathbf{then} & F := 1 \\ & \mathbf{else} & \underbrace{(I := I - 1;\ \mathbf{call}\ ;\ I := I + 1;\ F := I * F)}_{s} \end{array}$$

$$\frac{\dfrac{H \vdash \{I = 0\}\ F := 1\ \{F = I!\} \qquad \dfrac{H \vdash \{I \neq 0\}\ I := I - 1\ \{\mathbf{true}\ \}\ \overset{H}{\mathbf{call}}\ \{F = I!\}\ I := I + 1\ \{F = (I-1)!\}\ F = I * F\ \{F = I!\}}{H \vdash \{I \neq 0\}\ s\ \{F = I!\}}}{H \vdash \{\mathbf{true}\ \}\ body\ \{F = I!\}}}{\vdash \{\mathbf{true}\ \}\ \mathbf{call}\ \{F = I!\}}$$

Example:

$$\begin{array}{lll} body = \mathbf{if}\ I = 0 & \mathbf{then} & \mathbf{skip} \\ & \mathbf{else} & (I := I - 1;\ \mathbf{call}\ I := I + 1) \end{array}$$

$$\frac{\dfrac{H \vdash \{I = i \wedge I = 0\}\ \mathbf{skip}\ \{I = i\} \qquad H \vdash \{I = i\}\ I := I - 1\ \{I = i - 1\}\ \overset{?}{\mathbf{call}}\ \{I = i - 1\}\ I := I + 1\ \{I = i\}}{H \vdash \{I = i\}\ body\ \{I = i\}}}{\vdash \underbrace{\{I = i\}\ \mathbf{call}\ \{I = i\}}_{H}}$$

[20] (Hypothesis)

NB: i does not occur in the program. i is an auxiliary variable – remembers pre-values of I. Need to adapt the value of auxiliary variables.

A specification of addition?

$$\{\ \mathbf{true}\ \}\ X := 0;\ Y := 0;\ Z := 0\ \{\ Z = X + Y\ \}$$

With auxiliary variables:

$$\{X = x \wedge Y = y\} \quad Z := X + Y \quad \{Z = x + y\}$$

where x and y are auxiliary variables.

⌈ Alternative:

$$\{\ \mathbf{true}\ \} \qquad \{\ Z = X_{pre} + Y_{pre}\ \} \quad ⌋$$

Assertions = Predicates of (State of program variables × state of auxiliary variables)
Convention: auxiliary variable (i) = copy of program variable (I)
⇒ use Σ for both program and auxiliary variables

$$\text{Formally: Assn} = \underbrace{\Sigma}_{progr.\ var.} \times \underbrace{\Sigma}_{aux.\ var.} \rightarrow T$$

Convention: τ represents state of auxiliary variables.

Problem:

$$?\frac{\{I = i\}\ \mathbf{call}\ \{I = i\}}{\{I = i-1\}\ \mathbf{call}\ \{I = i-1\}} \tag{11}$$

New consequence rule

$$\frac{H \vdash \{P'\}\ s\ \{Q'\}}{H \vdash \{P\}\ s\ \{Q\}} \text{ if } \forall\sigma,\sigma'.\overbrace{\big(\forall\tau.P'(\sigma,\tau) \Rightarrow Q'(\sigma',\tau)\big)}^{\approx\models\{P'\}\ s\ \{Q\}} \Rightarrow \overbrace{\big(\forall\tau.P(\sigma,\tau) \Rightarrow Q(\sigma',\tau)\big)}^{\approx\models\{P\}\ s\ \{Q\}}$$

τ is universally quantified!

Validity with auxiliary variables:
$\models \{P\}\ s\ \{Q\} \equiv \forall\sigma,\sigma'.\langle s,\sigma\rangle \rightarrow \sigma' \Rightarrow \big(\forall\tau.P(\sigma,\tau) \Rightarrow Q(\sigma',\tau)\big)$

Solution for (11) by new consequence rule because

$$\forall \underbrace{I}_{\sigma}, \underbrace{I'}_{\sigma'} . \underbrace{(\forall \underbrace{i}_{\tau} . I = i \rightarrow I' = i)}_{\text{instantiate with } i-1} \Rightarrow (\forall i. I = i-1 \Rightarrow I' = i-1) \quad \checkmark$$

Old consequence rule is a special case of the new one:
Assume old side conditions:

1. $\forall\sigma,\tau.P(\sigma,\tau) \Rightarrow P'(\sigma,\tau)$
2. $\forall\sigma,\tau.Q'(\sigma,\tau) \Rightarrow Q(\sigma,\tau)$

Prove side condition of new consequence rule:
Assume (3) $\forall\tau.P'(\sigma,\tau) \Rightarrow Q'(\sigma',\tau)$.
Prove $\forall\tau.P(\sigma,\tau) \Rightarrow Q(\sigma',\tau)$

$$P(\sigma,\tau) \overset{(1)}{\Rightarrow} P'(\sigma,\tau) \overset{(3)}{\Rightarrow} Q'(\sigma',\tau) \overset{(2)}{\Rightarrow} Q(\sigma',\tau)$$

Soundness of new rule:
Assume $\models \{P'\}\ s\ \{Q'\} \Leftrightarrow \forall \sigma, \sigma'. \langle s, \sigma \rangle \rightarrow \sigma' \Rightarrow \underbrace{(\forall \tau. P'(\sigma, \tau) \Rightarrow Q'(\sigma', \tau))}_{(B)}$

Prove $\models \{P\}\ s\ \{Q\}$
Assume $\langle s, \sigma \rangle \rightarrow \sigma' \quad (A)$
Prove $\underbrace{\forall \tau. P(\sigma, \tau) \Rightarrow Q(\sigma', \tau)}_{(C)}$
$(A) \Rightarrow (B) \Rightarrow (C)$

Soundness

Validity of triples under assumptions:
$T_1 \models T_2 \equiv (\models T_1 \Rightarrow \models T_2)$

Theorem: $H \vdash \{P\}\ s\ \{Q\} \Rightarrow H \models \{P\}\ s\ \{Q\}$

Proof: "Validity is preserved by every rule."

- for basic constructs: as usual
- for new consequence rule: see above
- for assumption rule: ✓
- for call rule: $\dfrac{\{P\}\ \textbf{call}\ \{Q\} \vdash \{P\}\ body\ \{Q\}}{\vdash \{P\}\ \textbf{call}\ \{Q\}}$

  01/30/2002

  IH: $\{P\}\ \textbf{call}\ \{Q\} \models \{P\}\ body\ \{Q\}$
  i.e. $\models \{P\}\ \textbf{call}\ \{Q\} \rightarrow \models \{P\}\ body\ \{Q\}$

  Goal: $\{P\}\ \textbf{call}\ \{Q\}$ iff. $\{P\}\ body\ \{Q\}$
  (by definition of $\models$ and operational semantics)

???

Finer grained operational semantics:
$\langle s, \sigma \rangle \rightarrow_n \sigma'$ where n indicates the maximum recursion depth.

$\vec{n}$

Some of the rules:

$$\frac{\langle body, \sigma \rangle \rightarrow_n \sigma'}{\langle \textbf{call}\ , \sigma \rangle \rightarrow_{n+1} \sigma'}$$

$$\frac{\langle s_1, \sigma_1 \rangle \rightarrow_n \sigma_2 \qquad \langle s_2, \sigma_2 \rangle \rightarrow_n \sigma_3}{\langle s_1; s_2, \sigma_1 \rangle \rightarrow_n \sigma_3}$$

$$\models_n \{P\}\ s\ \{Q\} \equiv \forall \sigma, \sigma'. \langle s, \sigma \rangle \rightarrow_n \sigma' \Rightarrow (\forall \tau. P(\sigma, \tau) \Rightarrow Q(\sigma', \tau))$$
$$T_1 \models_n T_2 \equiv \models_n T \Rightarrow \models_n T_2$$

Theorem: $C \vdash \{P\}\ s\ \{Q\} \Rightarrow \underbrace{\forall n. C \models_n \{P\}\ s\ \{Q\}}$

Corollary (Proof needs theorems about the relationship of $\langle s, \sigma \rangle \rightarrow\rightarrow \sigma'$ and $\langle s, \sigma \rangle \rightarrow_n \sigma$

Proof: by induction on $C \models \{P\}\ s\ \{Q\}$
call : IH: $\forall n.\{P\}$ **call** $\{Q\} \models_n \{P\}\ body\ \{Q\}$
i.e. $\models_n \{P\}$ **call** $\{Q\} \rightarrow \models_n \{P\}\ body\ \{Q\}$
Goal: $\forall n. \models_n \{P\}$ **call** $\{Q\}$
Proof by induction on n:

1. $n = 0$ ✓
   ($\models_0 \{P\}$ **call** $\{Q\}$ is always true because $\langle \textbf{call}, \sigma\rangle \rightarrow_0 \sigma'$ is always false)

2. $n \rightsquigarrow n+1$

$$\begin{array}{rcl} IH_2 & \Rightarrow & \models_n \{P\}\ \textbf{call}\ \{Q\} \\ IH_1 & \Rightarrow & \models_n \{P\}\ body\ \{Q\} \\ & \Rightarrow & \models_{n+1} \{P\}\ \textbf{call}\ \{Q\} \end{array}$$

Completeness

$MGT(s)$[21] $= \{\lambda\sigma\tau.\tau = \sigma\}\ s\ \{\lambda\sigma'\tau.\langle s,\tau\rangle \rightarrow \sigma'\}$

$\models MGT(s) : \langle s,\sigma\rangle \rightarrow \sigma' \Rightarrow \underbrace{(\forall\tau.\tau = 0 \Rightarrow \langle s,\tau\rangle \rightarrow \sigma')}_{\langle s,\sigma\rangle \rightarrow \sigma'}$

Lemma: $\vdash MGT\ s$ (for all s) implies completeness.

Proof: $\dfrac{\vdash MGT\ s}{\vdash \{P\}\ s\ \{Q\}}$ because $\overbrace{(\forall\tau.\tau = \sigma \Rightarrow \langle s,\tau\rangle \rightarrow \sigma')}^{\langle s,\sigma\rangle \rightarrow \sigma'} \Rightarrow (\forall\tau.P(\sigma,\tau) \Rightarrow Q(\sigma',\tau))$

$\models \{P\}\ s\ \{Q\} \equiv \forall\sigma,\sigma'.\langle s,\sigma\rangle \rightarrow \sigma' \Rightarrow (\forall\tau.P(\sigma,\tau) \Rightarrow Q(\sigma',\tau))$
i.e. $\models \{P\}\ s\ \{Q\}$ implies side condition of new consequence rule.

Lemma: (1) $H \vdash MGT$ **call** $\Rightarrow H \vdash MGT\ s$

Now:

$$\begin{array}{rcl} & MGT\ \textbf{call} \vdash MGT\ \textbf{call} & \text{by assumption rule} \\ \Rightarrow & MGT\ \textbf{call} \vdash MGT\ body & \text{by Lemma (1)} \\ \Rightarrow & \vdash MGT\ \textbf{call} & \text{by } \textbf{call} \text{ rule} \\ \Rightarrow & \vdash MGT\ s & \text{by Lemma (1)} \end{array}$$

Proof: of Lemma (1) by induction on s.

- $s =$ **call** : ✓
- $s =$ **while** b **do** s_0

$$\uparrow \text{conseq.} \frac{H \vdash \{\lambda\sigma\tau.\tau = \sigma\}\ s\ \{\lambda\sigma'\tau.\langle s,\tau\rangle \rightarrow \sigma'\}}{H \vdash \{\lambda\sigma\tau.I(\sigma,\tau)\}\ s\ \{\lambda\sigma'\tau.I(\sigma',\tau) \wedge \neg\mathcal{B}[b]\sigma'\}}$$

$$I(\sigma,\tau) = (\tau,\sigma) \in \{(\sigma,\sigma') \mid \langle s_0,\sigma\rangle \rightarrow \sigma' \wedge \mathcal{B}[b]\sigma\}^*$$

[21] **M**ost **G**eneral **T**riple

= a finite number of iterations of s_0 take τ to σ. Side condition provable by induction on $*$.

5.10 A Hoare Logic for Time

Aim: $\{A\}\ s\ \{e \Downarrow B\} \approx$ execution of s takes time $O(e)$.

Plan:

1. operational semantics with exact time
2. timed triples
3. Hoare Logic for timed triples
4. Examples
5. Soundness

5.10.1 Operational Semantics with time

$$\mathcal{TA} : Aexp \to \mathbb{N}$$

Time for evaluating an $Aexp$. Machine model: unit cost for all arithmetic operations.

Definition of $\mathcal{TA}$:

$$\begin{aligned} & \mathcal{TA}(a_1\ aop\ a_2) = 1 + \mathcal{TA}(a_1) + \mathcal{TA}(a_2) \\ & \dots \\ \Rightarrow\ & \mathcal{TA}(a) = \text{size of } a \text{ (as a tree)} \end{aligned}$$

Similarly: $\mathcal{TB} : Bexp \to \mathbb{N}$

For statements: $\langle s, \sigma \rangle \to^n \sigma'$

$$\begin{array}{lll} \langle \mathbf{skip}\ , \sigma \rangle & \to^1 & \sigma \\ \langle x := a, \sigma \rangle & \to^{\mathcal{TA}(a)} & \sigma[x \mapsto \mathcal{A}[a]\sigma] \end{array}$$

$$\frac{\langle s_1, \sigma_1 \rangle \to^{n_1} \sigma_2 \qquad \langle s_2, \sigma_2 \rangle \to^{n_2} \sigma_3}{\langle s_1 ; s_2, \sigma \rangle \to^{n_1+n_2} \sigma_3}$$

$$\frac{\mathcal{B}[b]\sigma = \text{tt} \qquad \langle s_1, \sigma \rangle \to^n \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ s_1, \sigma \rangle \to^{n+\mathcal{TB}(b)} \sigma'}$$

$$\frac{\mathcal{B}[b]\sigma = \text{ff}}{\langle w, \sigma \rangle \to^{\mathcal{TB}(b)} \sigma} \qquad \frac{\mathcal{B}[b]\sigma = \text{tt} \qquad \langle s, \sigma \rangle \to^m \sigma' \qquad \langle w, \sigma' \rangle \to^n \sigma''}{\langle w, \sigma \rangle \to^{m+n+\mathcal{TB}(b)} \sigma''}$$

5.10.2 Timed Hoare triples

$\models \{A\}\ s\ \{e \Downarrow B\} \equiv$ there is some $k > 0$ ($k \in \mathbb{N}$ or $\mathbb{R}$) such that for all σ: if $A(\sigma)$ then there is some σ' and some n such that $\langle s, \sigma\rangle \longrightarrow^n \sigma'$ and $n \leq k * \mathcal{A}[e]\sigma$ and $B(\sigma')$.

NB: e a function of the *initial* state.

02/04/2002

Q: How to prove $\vdash \{I = J\}$ **call** $\{I = J\}$ where $body =$ **if** $I = 0$ **then skip else** $(I := I - 1;$ **call** $; I := I + 1)$

1. Prove $\text{conseq}\dfrac{\vdash \{I = i \wedge J = j\}\ \textbf{call}\ \{I = i \wedge J = j\}}{\vdash \{I = J\}\ \textbf{call}\ \{I = J\}}$

   $(\forall i, j. I = i \wedge J = j \Rightarrow I' = i \wedge J' = j)$ (set $i = j$)
   $\Rightarrow (\forall i, j. I = J \Rightarrow I' = J')$

Hoare Logic with time:

$$\begin{array}{lcl} \models \{X = 42\} & \textbf{while } X \neq 0 \textbf{ do } X := X - 1 & \{\overset{1}{X} \Downarrow X = 0\} \\ \models \{X \geq 0\} & '' & \{X \Downarrow X = 0\} \\ \models \{\textbf{ true }\} & '' & \text{no possible assertion} \end{array}$$

Finer grained definition of $\models$ with separated auxiliary variables (τ)

$$\models \{A\}\ s\ \{t \Downarrow B\} \equiv \exists k > 0. \forall \sigma, \tau. A(\sigma, \tau) \Rightarrow \exists \sigma', n. \langle s, \sigma\rangle \longrightarrow^n \sigma' \wedge n \leq k * \mathcal{A}[t](\sigma, \tau) \wedge B(\sigma', \tau)$$

5.10.3 Rules of Hoare Logic for Time

$$\begin{array}{ccc} \{B[a/x]\} & x := a & \{1 \Downarrow B\} \\ \{A\} & \textbf{skip} & \{1 \Downarrow A\} \end{array}$$

$$\frac{\{A_1\}\ s_1\ \{t_1 \Downarrow A_2\} \qquad \{A_2\}\ s_2\ \{t_2 \Downarrow A_3\}}{\{A_1\}\ s_1; s_2\ \{t_1 + t_2 \Downarrow A_3\}}$$

Example:

$$\lightning\ \frac{\{Y \geq 0\}\ X := Y\ \{1 \Downarrow X \geq 0\} \qquad \{X \geq 0\}\ w\ \{X \Downarrow X = 0\}}{\{Y \geq 0\}\ X := Y;\ \underbrace{\textbf{while } X \geq 0 \textbf{ do } X := X - 1}_{w}\ \{1 + X \Downarrow X = 0\}}$$

Problem: t_2 refers to the intermediate state, not to the initial state.

New rule:

$$\frac{\{A_1 \wedge t_2' = u\}\ s_1\ \{t_1 \Downarrow A_2 \wedge t_2 \leq u\} \qquad \{A_2\}\ s_2\ \{t_2 \Downarrow A_3\}}{\{A_1\}\ s_1; s_2\ \{t_1 + t_2 \Downarrow A3\}}$$

where u is some new auxiliary variable.

Example:

$$\frac{\{Y \geq 0 \wedge \overset{t_2'}{Y} = u\}\ X := Y\ \{\overset{t_1}{1} \Downarrow X \geq 0 \wedge \overset{t_2}{X} \leq u\} \qquad \{X \geq 0\}\ w\ \{\overset{t_2}{X} \Downarrow X \geq 0\}}{\{Y \geq 0\}\ X := Y;\ \underbrace{\textbf{while } X \geq 0 \textbf{ do } X := X - 1}_{w}\ \{1 + Y \Downarrow X \geq 0\}}$$

if -rule:

$$\frac{\{A \wedge b\}\ s_1\ \{t \Downarrow B\} \qquad \{A \wedge \neg B\}\ s_2\ \{t \Downarrow B\}}{\{A\}\ \textbf{if}\ b\ \textbf{then}\ \ldots\ \{t \Downarrow B\}}$$

If s_1 and s_2 have different complexities: use consequence rule:

$$\frac{A \Rightarrow (A' \wedge \exists k > 0.t' \leq k \cdot t) \qquad \{A'\}\ s\ \{t' \Downarrow B'\} \qquad B' \Rightarrow B}{\{A\}\ s\ \{t \Downarrow B\}}$$

Example:

$$\text{conseq}\,\frac{\text{conseq}\,\dfrac{\{A\}\ s\ \{1 \Downarrow B\}}{\{A\}\ s\ \{2 \Downarrow B\}}}{\{A\}\ s\ \{1 \Downarrow B\}}$$

$$\frac{\{X \geq 0\}\ s\ \{1 \Downarrow B\}}{\{''\}\ s + \{X \Downarrow B\}}$$

$$\frac{\{A(n+1) \wedge t' = u\}\ s\ \{t_s \Downarrow A(n) \wedge t \leq u\}}{\{\exists n.A(n)\}\ w\ \{t \Downarrow A(0)\}}$$

$$if \qquad \begin{array}{c} A(0) \Rightarrow \neg b \wedge t \geq 1 \\ A(n+1) \Rightarrow b \wedge t \geq t' + t_s \end{array}$$

$$\frac{\{\ldots \wedge t' = u\}\ s\ \{t_s \Downarrow \ldots \wedge t \leq u\} \qquad \{\ldots\}\ w\ \{t \Downarrow \ldots\}}{\{\ldots\}\ s;w\ \{t' + t_s \Downarrow \ldots\}}$$

Example:

$$\text{cnsq}\,\frac{\dfrac{\text{cnsq}\,\dfrac{\dfrac{\{\overbrace{I = n+1 \wedge I = u}^{A_2} \wedge 1 = v\}\ S := S+I\ \{1 \Downarrow A_2 \wedge 1 \leq v\} \qquad \{A_2\}\ I := I-1\ \{1 \Downarrow I = n \wedge I+1 \leq u\}}{\{I = n+1 \wedge \overbrace{I}^{t'} = u\}\ S := S+I;\ I := I-1\ \{2 \Downarrow I = n \wedge I+1 \leq u\}}}{\{I = n+1 \wedge \overbrace{I}^{t'} = u\}\ S := S+I;\ I := I-1\ \{1 \Downarrow I = n \wedge I+1 \leq u\}}}{\{\exists n.I = n\}\ w\ \{I+1 \Downarrow \textbf{true}\}}}{\{I \geq 0\}\ \textbf{while}\ I \neq 0\ \textbf{do}\ (S := S+I;\ I := I-1)\ \{\underbrace{I+1}_{t} \Downarrow \textbf{true}\}}$$

$$\begin{array}{c} I = 0 \Rightarrow \neg(I \neq 0) \wedge I+1 \geq 1 \\ I = n+1 \Rightarrow I \neq 0 \wedge I+1 \geq I+1 \quad \checkmark \end{array}$$

$\mathcal{EOF}$

References

[WINSKEL93] The Formal Semantics of Programming Languages. An Introduction. Glynn Winskel, 1993.

[NIELSON99] Semantics with Applications: A Formal Introduction. Hanne Riis Nielson, Flemming Nielson, 1999. http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html

[LECTURE] Homepage of the lecture: http://www4.in.tum.de/lehre/vorlesungen/semantik/WS0102/