Programming Languages and Compiler Design

Programming Language Semantics
Compiler Design Techniques

Yassine Lakhnech & Laurent Mounier

{lakhnech,mounier}@imag.fr

http://www-verimag.imag.fr/˜lakhnech

http://www-verimag.imag.fr/˜mounier.

Master of Sciences in Informatics at Grenoble (MoSIG)

Grenoble Universités

(Université Joseph Fourier, Grenoble INP)

Teachers

Lecture sessions

- Yassine Lakhnech
- Laurent Mounier

Exercice sessions (TD)

- Yliés Falcone

Web page:
http://www-verimag.imag.fr/˜lakhnech

Office location:
Verimag Lab, CTL building, av. de Vignate, Gières

Course objectives

- Programming languages, and their description
- General compiler architecture
- Some more detailed compiler techniques

References

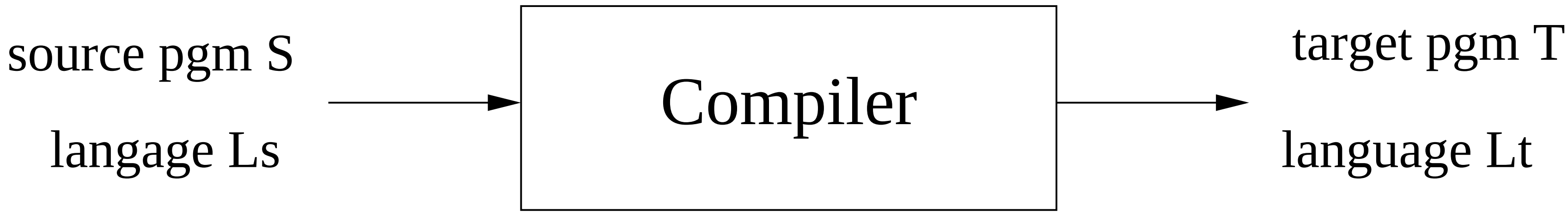
A. Aho, R. Sethi and J. Ullman Compilers: Principles, techniques and tools InterEditions, 1989

H. R. Nielson and F. Nielson. Semantics with Applications: An Appetizer. Springer, March 2007. ISBN 978-1-84628-691-9

W. Waite and G. Goos. Compiler Construction Springer Verlag, 1984

R. Wilhelm and D. Maurer. Compilers - Theory, construction, generation Masson 1994

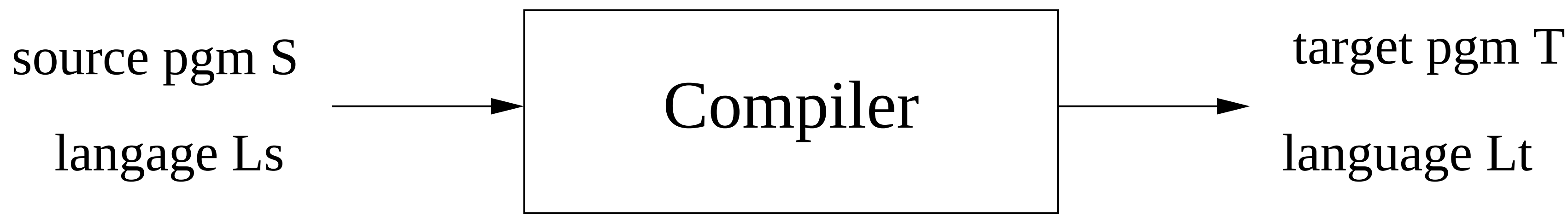
Compiler: what do we expect ?



Expected Properties ?

-
-
-

Compiler: what do we expect ?



Expected Properties ?

- correctness:
  execution of T should preserve the **semantics** of S
- efficiency:
  T should be optimized w.r.t. some execution resources (time, memory, energy, etc.)
- "user-friendness": errors in S should be accurately reported
- completeness: any correct Ls-program should be accepted

Many programming langage paradigms . . .

Imperative langages

Fortran, Algol-xx, Pascal, C, Ada, Java, etc
control structure, explicit memory assignment, expressions

Functional languages

ML, CAML, LISP, Scheme, etc
term reduction, function evaluation

Object-oriented langages

Java, Ada, Eiffel, ...
objets, classes, types, heritage, polymorphism, . . .

Logical languages

Prolog
resolution, unification, predicate calculus, . . .

etc.

. . . and many architectures to target !

- CISC
- RISC
- VLIW, multi-processor architectures
- dedicated processors (DSP, . . .).
- etc.

We will mainly focus on:

Imperative languages

- data structures
  - basic types (integers, characters, pointers, etc)
  - user-defined types (enumeration, unions, arrays, . . .)
- control structures
  - assignements
  - iterations, conditionals, sequence
  - nested blocks, sub-programs

“Standart” general-purpose machine architecture: (e.g. ARM, iX86)

- heap, stack and registers
- arithmetic and logical binary operations
- conditional branches

Language Description

For a programming language P

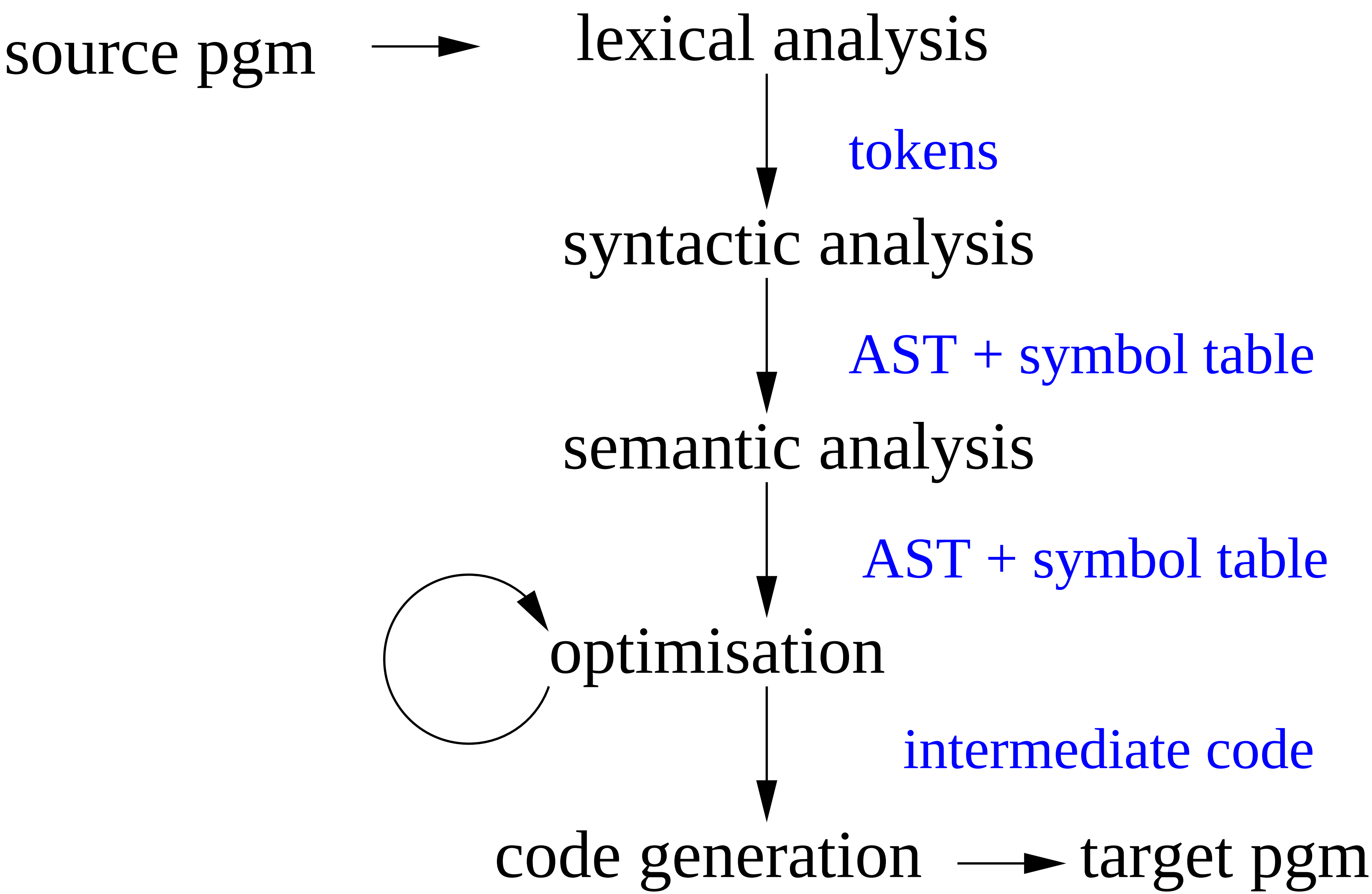
Lexicon L: words of P
$\longrightarrow$ a regular language over P alphabet

Syntax S: sentences of P
$\longrightarrow$ a context-free language over L

Static semantic (e.g., typing): “meaningful” sentences of P
$\longrightarrow$ subset of S, defined by inference rules or attribute grammars

Dynamic semantic: the meaning of P programs
$\longrightarrow$ transition relation, predicate transformers, partial functions

Compiler architecture



Lexical analysis

Input: sequence of characters

Output: sequence of lexical unit classes

1. compute the longest sequence $\in$ a given lexical class
   lexical classes (*token*): constants, identifiers, keywords, operators, separators, . . .
2. skip the comments
3. special token: error

Formal tool: **regular languages**

(deterministic) finite automata $\Leftrightarrow$ regular expressions

Syntactic Analysis (parsing)

Input: sequence of tokens

Output: abstract syntax tree (AST) + symbol table

1. syntactic analysis of the input sequence
2. AST construction (from a derivation tree)
3. insert the identifiers in a symbol table

Formal tools: **context-free languages** (CFG)
Pushdown automata, context-free grammars

deterministic pushdown automata $\Leftrightarrow$ **strict subset** of CFG

Semantic analysis

Input: : Abstract syntax tree (AST)

Output: : enriched AST

- name identification:
  $\longrightarrow$ bind **use-def** occurrences
- type verification and/or type inference

$\Rightarrow$ traversals and modifications of the AST

Code generation

Input: AST

Output: intermediate code, machine code

- based on a systematic translation functions f
- should ensure that:

$$\mathsf{Sem}_{\mathsf{source}}(\mathsf{P}) = \mathsf{Sem}_{\mathsf{cible}}(\mathsf{f}(\mathsf{P}))$$

- in practice: several intermediate code levels (to ease the optimisation steps)

Optimisation

Input/Output: intermediate code

- several criteria: execution time, size of the code, energy
- several optimization levels (source level vs machine level)
- several techniques:
  - data-flow analysis
  - abstract interpretation
  - typing systems
  - etc.

Programming language semantics

Motivation

Why do we need to study programming language semantics ?
Semantics is essential to:

- understand programming languages
- validate programs
- write program specifications
- write compilers (and program transformers)
- classify programming languages

Why do we need to formalise this semantics ?

Example: static vs. dynamic binding

```
Program Static_Dynamic
var a := 1;
proc p(x);
    begin
        a := x + 1; write(a)
    end;
proc q
    var a := 2;
    begin
        p(a)
    end;
```

Example: static vs. dynamic binding

```
Program Static_Dynamic
var a := 1;
proc p(x);
    begin
        a := x + 1; write(x)
    end;
proc q
    var a := 2;
    begin
        p(a)
    end;
```

what value is printed ?

Example: parameters

Program value_reference

var a;

proc $p(x)$;

begin

$a := x + 1; write(a); write(x)$

end;

begin

$a := 2; p(a); write(a)$

end;

Example: parameters

Program value_reference

var a;

proc $p(x)$;

begin

$x := x + 1; write(a); write(x)$

end;

begin

$a := 2; p(a); write(a)$

end;

What values are printed ?

Example: parameters

Program value_reference

var a;

proc $p(x)$;

begin

$x := x + 1; write(a); write(x)$

end;

begin

$a := 2; p(a); write(a)$

end;

2 3 2 , if call-by-value

3 3 3, if call-by-reference

Course overview

- Semantic styles
  - Natural operational semantic
  - Axiomatic semantics (Hoare logic)
  - Denotational semantics
- Languages considered:
  - imperative
  - functional

The **While** *language*

x : variable
S : statement
a : arithmetic expression
b : boolean expression

$$
\begin{array}{rcl}
S & ::= & x := a \mid \text{skip} \mid S_1 ; S_2 \mid \\
 & & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & & \text{while } b \text{ do } S \text{ od}
\end{array}
$$

syntactic categories 1

- Numbers

$$n \in \textbf{Num} = \{0, \cdots, 9\}^+$$

- Variables

$$x \in \textbf{Var}$$

- Arithmetic expressions

$$\begin{array}{lll} a & \in & \textbf{Aexp} \\ a & := & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \end{array}$$

Syntactic categories 2

- Boolean expressions

$$b \in \textbf{Bexp}$$

$$b := \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

- Statements

$$S \in \textbf{Stm}$$

$$S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid$$
$$\text{if } b \text{ then } S_1 \text{ else } S_2$$
$$\text{while } b \text{ do } S \text{ od}$$

Concrete vs. abstract syntax

- The term $S_1; S_2$ represents the tree those root is $;$, left child is S_1 tree, and right child is S_2 tree.
- Parenthesis shall be used to avoid ambiguities. Example :

$$S_1, (S_2; S_3) \text{ and } (S_1; S_2); S_3$$

Semantic domains

Integers : $\mathbb{Z}$

Booleans : $\mathbb{B}$

States :

$$\textbf{State} = \textbf{Var} \rightarrow \mathbb{Z}$$

Let $v \in \mathbb{Z}$. Then $\sigma[y \mapsto v]$ denotes state σ' such that:

$$\sigma'(x) = \begin{cases} \sigma(x) & \text{if } x \neq y \\ v & \text{otherwise} \end{cases}$$

Semantic functions 1

- Digits : integers

$$\mathcal{N} : \textbf{Num} \longrightarrow \mathbb{Z}$$

- Arithmetic expressions: for each state, a value in $\mathbb{Z}$

$$\mathcal{A} : \textbf{Aexp} \longrightarrow (\textbf{State} \longrightarrow \mathbb{Z})$$

Semantic functions 2

- Boolean expressions: for each state, a value in $\mathbb{B}$

$$\mathcal{B} : \textbf{Bexp} \longrightarrow (\textbf{State} \longrightarrow \mathbb{B})$$

- Statements:

$$\mathcal{S} : \textbf{Stm} \longrightarrow (\textbf{State} \stackrel{part.}{\longrightarrow} \textbf{State})$$

Arithmetic expressions semantics

$$\mathcal{N}(n_1 \cdots n_k) = \Sigma_{i=1}^{k} n_i \cdot 10^{k-i}$$

$$\begin{aligned}
\mathcal{A}[n]\sigma &= \mathcal{N}[n] \\
\mathcal{A}[x]\sigma &= \sigma(x) \\
\mathcal{A}[a_1 + a_2]\sigma &= \mathcal{A}[a_1]\sigma +_I \mathcal{A}[a_2]\sigma \\
\mathcal{A}[a_1 * a_2]\sigma &= \mathcal{A}[a_1]\sigma *_I \mathcal{A}[a_2]\sigma \\
\mathcal{A}[a_1 - a_2]\sigma &= \mathcal{A}[a_1]\sigma -_I \mathcal{A}[a_2]\sigma
\end{aligned}$$

The semantics of arithmetic expressions is inductively defined over their structure. It is a compositional semantics.

Boolean expressions semantics

Exercice : define the semantics of boolean expressions . . .

Various semantic styles

- Operational semantics tells how a program is executed. It helps to write interpreters or code generators.
- Axiomatic semantics allows to prove program properties.
- Denotational semantics describes the effect of program execution (from a given state), without telling how the program is executed.

Another important feature is *compositionality*: the semantics of a compound program is a function of the semantics of its components.

Operational Semantic

An operational semantics defines a transition system.
A transition system is given by:

$$(\Gamma, T, \rightarrow)\text{where}$$

- Γ is the configuration set.
- $T \subseteq \Gamma$ is the set of final configurations.
- $\rightarrow \subseteq \Gamma \times \Gamma$ is the transition relation.

Natural semantic 1

Goal: to describe how the result of a program execution is obtained.
Semantic defined by an inference system: axioms and rules.

$$(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$(\textsf{skip}, \sigma) \to \sigma$$

$$\frac{(S_1, \sigma) \to \sigma', \quad (S_2, \sigma') \to \sigma''}{(S_1; S_2, \sigma) \to \sigma''}$$

Natural semantic 2

If $\mathcal{B}[b]\sigma = \textbf{tt}$ then

$$\frac{(S_1, \sigma) \rightarrow \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \rightarrow \sigma'}$$

If $\mathcal{B}[b]\sigma = \textbf{ff}$ then

$$\frac{(S_2, \sigma) \rightarrow \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \rightarrow \sigma'}$$

If $\mathcal{B}[b]\sigma = \textbf{tt}$ then

$$\frac{(S, \sigma) \rightarrow \sigma', \quad (\text{while } b \text{ do } S \text{ od }, \sigma') \rightarrow \sigma''}{(\text{while } b \text{ do } S \text{ od }, \sigma) \rightarrow \sigma''}$$

Natural semantic 3

If $\mathcal{B}[b]\sigma = \textbf{tt}$ then

$$\frac{(S,\sigma) \rightarrow \sigma', \quad (\text{while } b \ \text{ do } S \text{ od }, \sigma') \rightarrow \sigma''}{(\text{while } b \ \text{ do } S \text{ od }, \sigma) \rightarrow \sigma''}$$

If $\mathcal{B}[b]\sigma = \textbf{ff}$ then

$$(\text{while } b \ \text{ do } S \text{ od }, \sigma) \rightarrow \sigma$$

Derivation tree

- Leaves correspond to axioms
- Nodes corresponds to inference rules.

Example : what is the semantic of:

1. $x := 2;$ while $x > 0$ do $x := x - 1$ od
2. $x := 2;$ while $x > 0$ do $x := x + 1$ od

The natural semantics is deterministic

Théorème for all statement $S \in$ **Stm**, for all states σ, σ' and σ'':

1. If $(S, \sigma) \rightarrow \sigma'$ and $(S, \sigma) \rightarrow \sigma''$ then $\sigma' = \sigma''$.
2. If $(S, \sigma) \rightarrow \sigma'$ then it does not exist an infinite derivation tree.

Preuve by induction on the structure of the derivation tree. $\square$

The semantic function $\mathcal{S}_{ns}$

$$\mathcal{S}_{ns}[S]\sigma = \begin{cases} \sigma' & ; \quad \text{If } (S,\sigma) \rightarrow \sigma' \\ \text{undef} & ; \quad \text{otherwise} \end{cases}$$

Blocks and procedures

Blocks and variable declarations

$$
\begin{array}{lcl}
S & \in & \textbf{Stm} \\
S & ::= & x := a \mid \text{skip} \mid S_1 ; S_2 \mid \\
 & & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & & \text{while } b \text{ do } S \text{ od } \mid \text{begin } D_V ; S \text{ end}
\end{array}
$$

The syntactic category **Dec**$_V$

$$D_V ::= \text{var } x := a;\ D_V \mid \epsilon$$

Example

$$
\begin{array}{ll}
\textsf{begin} & \textsf{var } y := 1; \\
 & (x := 1; \\
 & \textsf{begin var } x := 2; y := x + 1 \textsf{ end} \\
 & x := y + x) \\
\textsf{end} &
\end{array}
$$

Example

$$
\begin{array}{ll}
\textsf{begin} & \textsf{var } y := 1; \\
 & (x := 1; \\
 & \textsf{begin var } x := 2; y := x + 1 \textsf{ end} \\
 & x := y + x) \\
\textsf{end} &
\end{array}
$$

Questions:

1. Are the declaration actives during declaration execution ?
2. which order to choose when executing the declarations?
3. How to restore the initial state?

Natural operational semantics

Notation:

- DV(D_V) denotes the set of variables declared in D_V.
- $\sigma'[X \mapsto \sigma] = \lambda x \cdot$ if $x \in X$ then $\sigma(x)$ else $\sigma'(x)$.

To define the semantics we define a transition system for the declarations and a transition system for the statements.

Déclarations :

Configurations of the form (D_v, σ) or σ.

$$\frac{(D_V, \sigma[x \mapsto \mathcal{A}[a]\sigma]) \rightarrow_D \sigma'}{(\textsf{var}\ \ x := a;\ D_V, \sigma) \rightarrow_D \sigma'}$$

$$(\epsilon, \sigma) \rightarrow_D \sigma$$

Transition rules for the blocks

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (S, \sigma') \rightarrow \sigma''}{(\text{begin } D_V ; S \text{ end}, \sigma) \rightarrow \sigma''}$$

Transition rules for the blocks

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (S, \sigma') \rightarrow \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \rightarrow \sigma''[\mathsf{DV}(D_V) \mapsto \sigma]}$$

Transition rules for the blocks

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (S, \sigma') \rightarrow \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \rightarrow \sigma''[\mathsf{DV}(D_V) \mapsto \sigma]}$$

Rule for sequential composition

$$\frac{(S_1, \sigma) \rightarrow \sigma' \quad (S_2, \sigma') \rightarrow \sigma''}{(S_1; S_2, \sigma') \rightarrow \sigma''}$$

Procedure

$$
\begin{array}{lcl}
S & \in & \textbf{Stm} \\
S & ::= & x := a \mid \textsf{skip} \mid S_1; S_2 \mid \\
 & & \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \\
 & & \textsf{while } b \textsf{ do } S \textsf{ od } \mid \textsf{begin } D_V\ D_P; S \textsf{ end} \mid \textsf{call } p \\
D_V & ::= & \textsf{var } x := a;\ D_V \mid \epsilon \\
D_P & ::= & \textsf{proc } p \textsf{ is } S;\ D_P \mid \epsilon
\end{array}
$$

Procedure declarations are a syntactic category called **Dec**$_P$.

Example

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
                proc p is x := x + 1;
                call q; y := x;
        end;
end
```

Example 2

Dynamic binding for variables and procedures.

begin var $x := 0;$
proc p is $x := x * 2;$
proc q is call $p;$
begin var $x := 5;$
proc p is $x := x + 1;$
call $q; y := x;$
end;
end

Example 2

Dynamic binding for variables and procedures.

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call p; y := x;
        end;
end
```

Example 2

Dynamic binding for variables and procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call p;

begin var $x := 5;$

proc p is $x := x + 1;$

$x := x + 1; y := x;$

end;

end

Exemple 3

Dynamic binding for the variables and static binding for the procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call $p;$

begin var $x := 5;$

proc p is $x := x + 1;$

call $q; y := x;$

end;

end

Exemple 3

Dynamic binding for the variables and static binding for the procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call $p;$

begin var $x := 5;$

proc p is $x := x + 1;$

call $p; y := x;$

end;

end

Exemple 3

Dynamic binding for the variables and static binding for the procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call $p;$

begin var $x := 5;$

proc p is $x := x + 1;$

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

end;

end

Exemple 4

Static binding for variables and procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call $p;$

begin var $x := 5;$

proc p is $x := x + 1;$

call $q; y := x;$

end;

end

Exemple 4

Static binding for variables and procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call $p;$

begin var $x := 5;$

proc p is $x := x + 1;$

call $p; y := x;$

end;

end

Exemple 4

Static binding for variables and procedures.

begin var $x := 0;$

proc p is $x := x * 2;$

proc q is call $p;$

begin var $x := 5;$

proc p is $x := x + 1;$

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

end;

end

Semantics: dynamic bindings

A state associates a value (an integer) to a variable.
An *environment* associates a value to a procedure.

$$\textbf{Env}_P = \textbf{Pname} \xrightarrow{part.} \textbf{Stm}$$

Configurations: $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.

Transition rules

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (\mathsf{upd}(env, D_P), S, \sigma') \rightarrow \sigma''}{(env, \mathsf{begin}\ D_V\ D_P; S\ \mathsf{end}, \sigma) \rightarrow \sigma''[\mathsf{DV}(D_V) \mapsto \sigma]}$$

where

- $\mathsf{upd}(env, \epsilon) = env$ et
- $\mathsf{upd}(env, \mathsf{proc}\ p\ \mathsf{is}\ S; D_P) = \mathsf{upd}(env[p \mapsto S], D_P)$.

$$\frac{(env, env(p), \sigma) \rightarrow \sigma'}{(env, \mathsf{call}\ p, \sigma) \rightarrow \sigma'}$$

Rule for sequential composition

$$\frac{(env, S_1, \sigma) \rightarrow \sigma' \quad (env, S_2, \sigma') \rightarrow \sigma''}{(env, S_1; S_2, \sigma) \rightarrow \sigma''}$$

Semantics: static binding for procedures

$$\textbf{Env}_P = \textbf{Pname} \stackrel{part.}{\longrightarrow} \textbf{Stm} \times \textbf{Env}_P$$

Configurations: $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup$ **State.**

- upd$(env, \epsilon) = env$ and
- upd$(env,$ proc p is $S; D_P) =$ upd$(env[p \mapsto (S, env)], D_P)$.

begin var $x := 2$;
proc p is $x := 0$;
proc q is begin $x := 1$; (proc p is call p); call p end;
call q
end

Semantics: static binding for procedures

$$\textbf{Env}_P = \textbf{Pname} \stackrel{part.}{\longrightarrow} \textbf{Stm} \times \textbf{Env}_P$$

Configurations: $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup$ **State.**

- $\text{upd}(env, \epsilon) = env$ and
- $\text{upd}(env, \text{proc } p \text{ is } S; D_P) = \text{upd}(env[p \mapsto (S, env)], D_P)$.

begin var $x := 2$;
proc p is $x := 0$;
proc q is begin $x := 1$; (proc p is call p); call p end;
call q
end

Semantics: static binding for procedures

$$\textbf{Env}_P = \textbf{Pname} \overset{part.}{\longrightarrow} \textbf{Stm} \times \textbf{Env}_P$$

Configurations: $(\textbf{Env}_P \times \textbf{Stm} \times \textbf{State}) \cup \textbf{State}$.

- upd$(env, \epsilon) = env$ and
- upd$(env,$ proc p is $S; D_P) =$ upd$(env[p \mapsto (S, env)], D_P)$.

begin var $x := 2$;
proc p is $x := 0$;
proc q is begin $x := 1$; (proc p is call p); call p end;
call q
end

Transition rules

Two alternatives:

$$[\text{call}] \quad \frac{(env', S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \text{ où } env(p) = (S, env').$$

$$[\text{call}_{rec}] \quad \frac{(env'[p \mapsto (S, env')], S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \text{ où } env(p) = (S, env').$$

Transition rules

Two alternatives:

$$[\text{call}] \quad \frac{(env', S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \text{ où } env(p) = (S, env').$$

$$[\text{call}_{rec}] \quad \frac{(env'[p \mapsto (S, env')], S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \text{ où } env(p) = (S, env').$$

Rule for sequential composition

$$\frac{(env, S_1, \sigma) \rightarrow \sigma' \quad (env, S_2, \sigma') \rightarrow \sigma''}{(env, S_1; S_2, \sigma) \rightarrow \sigma''}$$

Semantics: static bindings

States are replaced by a symbol table and a memory:

- a symbol table associates a variable (an identifier) with a memory address.
- a memory associates an address with a value.

Symbol table: variable environment:

$$\textbf{Env}_V = \textbf{Var} \stackrel{part.}{\longrightarrow} \textbf{Loc} = \mathbb{Z}$$

Memory : $\textbf{Store} = \textbf{Loc} \stackrel{part.}{\longrightarrow} \mathbb{Z}$
We denote by $\mathsf{new}(sto)$ the smallest integer n such that $sto(n)$ is not defined.
Intuition: function state corresponds to $sto \circ env_V$.

Configurations

- Variable declarations:
  $(\textbf{Dec}_V \times Envv \times \textbf{Store}) \cup (Envv \times \textbf{Store})$.
- $\textbf{Env}_P = \textbf{Pname} \stackrel{part.}{\longrightarrow} \textbf{Stm} \times \textbf{Env}_V \times \textbf{Env}_P$.
  - $\text{upd}(env_V, env_P, \epsilon) = env_P$ and
  - $\text{upd}(env_V, env_P, \text{proc } p \text{ is } S; D_P) =$
    $\text{upd}(env_V, env_P[p \mapsto (S, env_V, env_P)], D_P)$.
- Statements: $(\textbf{Env}_V \times \textbf{Env}_P \times \textbf{Stm} \times \textbf{Store}) \cup (\textbf{Env}_V \times \textbf{Store})$.

Semantic rules for variable declarations

$$\frac{(D_V, env_V[x \mapsto \mathsf{new}(sto)], sto[\mathsf{new}(sto) \mapsto v]) \rightarrow_D (env'_V, sto')}{(\mathsf{var}\ \ x := a;\ D_V, env_V, sto) \rightarrow_D (env'_V, sto')}$$

where $v = \mathcal{A}[a](sto \circ env_V)$.

$$(\epsilon, env_V, sto) \rightarrow_D (env_V, sto)$$

Semantic rules for statements

$$(env_V, env_P, x := a, sto) \rightarrow (env_V, sto[env_V(x) \mapsto \mathcal{A}[a](sto \circ env_V)])$$

$$[\text{call}] \quad \frac{(env'_V, env'_P, S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}.$$

$$[\text{call}_{rec}] \quad \frac{(env'_V, env'_P[p \mapsto (S, env'_V, env'_P)], S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}$$

où $env_P(p) = (S, env'_V, env'_P)$.

Semantic rules for statements

$$(env_V, env_P, x := a, sto) \rightarrow (env_V, sto[env_V(x) \mapsto \mathcal{A}[a](sto \circ env_V)])$$

$$[\text{call}] \quad \frac{(env'_V, env'_P, S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}.$$

$$[\text{call}_{rec}] \quad \frac{(env'_V, env'_P[p \mapsto (S, env'_V, env'_P)], S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}$$

où $env_P(p) = (S, env'_V, env'_P)$.

Rule for sequential composition

$$\frac{(env_V, env_P, S_1, \sigma) \rightarrow \sigma' \quad (env_V, env_P, S_2, \sigma') \rightarrow \sigma''}{(env_V, env_P, S_1; S_2, \sigma) \rightarrow \sigma''}$$

Correct Code Generation

- Define the operational semantics of an abstract machine. More specifically a stack machine.
- Specify a code generator for the **While** language by induction on the structure of programs.
- Use the operational semantics to prove correctness of the specified code generator.

The abstract machine AM

The operational semantics of the abstract machine AM is defined as a transition system whose configurations are triples consisting of:

- A list $instr_1, \cdots, instr_n$ of instructions. This is the code that has to be executed.
- A stack that is used to evaluate expressions.
- The memory of the machine, that is described as a mapping from variables to $\mathbb{Z}$.

The machine AM has neither registers nor accumulators. The head of the stack plays the role of an accumulator and the rest of the stack as registers.

The instruction set

Instruction	Effect
push-n,True,False	push the constant n,**tt**,**ff**
fetch(x) store(x)	push the current value of x pop the head of the stack and save it in x
add sub,mult.and,le,equal,neg branch(c_1,c_2) loop(c_1,c_2) noop	replace the head of the stack and the next element by their sum similar if the head of the stack is **tt** execute c_1 if it is **ff**, execute c_2, otherwise stop execute c_1, then if the head is **tt** execute c_2 followed by loop(c_1,c_2)) if it is **ff** stop skip

The transition relation

A program of the abstract machine is a sequence of instructions. The set of all programs is denoted **Code**.
A configuration of Am is a triple (c, p, m), where c is a program, p is the content of the stack in $(\mathbb{Z} \cup \mathbb{B})^*$ and m is a memory in **State**.
The transition relation $\triangleright$ is defined as follows :

$$
\begin{array}{lll}
(\text{push-n} \cdot c, p, m) & \triangleright (c, n \cdot p, m) & \\
(\text{True} \cdot c, p, m) & \triangleright (c, \textbf{tt} \cdot p, m) & \\
(\text{False} \cdot c, p, m) & \triangleright (c, \textbf{ff} \cdot p, m) & \\
(\text{fetch}(x) \cdot c, p, m) & \triangleright (c, m(x) \cdot p, m) & \\
(\text{store}(x) \cdot c, v \cdot p, m) & \triangleright (c, p, m[x \mapsto v]) & \text{if } v \in \mathbb{Z} \\
(\text{add} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 + v_2) \cdot p, m) & \text{if } v_1, v_2 \in \mathbb{Z} \\
(\text{sub} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 - v_2) \cdot p, m) & \text{if } v_1, v_2 \in \mathbb{Z}
\end{array}
$$

The transition relation cnt.

$$
\begin{array}{lll}
(\mathsf{mult} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 * v_2) \cdot p, m) & \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{le} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 \leq v_2) \cdot p, m) & \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{equal} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 = v_2) \cdot p, m) & \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{and} \cdot c, b_1 \cdot b_2 \cdot p, m) & \triangleright (c, (b_1 \wedge b_2) \cdot p, m) & \text{if } b_1, b_2 \in \mathbb{B} \\
(\mathsf{neg} \cdot c, b \cdot p, m) & \triangleright (c, (\neg b) \cdot p, m) & \text{if } b \in \mathbb{B} \\
(\mathsf{branch}(c_1, c_2) \cdot c, \mathbf{tt} \cdot p, m) & \triangleright (c_1 \cdot c, p, m) & \\
(\mathsf{branch}(c_1, c_2) \cdot c, \mathbf{ff} \cdot p, m) & \triangleright (c_2 \cdot c, p, m) & \\
(\mathsf{loop}(c_1, c_2) \cdot c, p, m) & \triangleright & \\
\multicolumn{3}{r}{(c_1 \cdot \mathsf{branch}(c_2 \cdot \mathsf{loop}(c_1, c_2), \mathsf{noop}) \cdot c, p, m)} \\
(\mathsf{noop} \cdot c, p, m) & \triangleright (c, p, m) &
\end{array}
$$

Some properties of AM

- The transition relation $\triangleright$ is deterministic:

  $(c,p,m) \triangleright (c_1,p_1,m_1) \wedge (c,p,m) \triangleright (c_2,p_2,m_2) \Rightarrow (c_1,p_1,m_1) = (c_2,p_2,$

  Proof by induction on the length of c.

- Extentionality of the code and the stack:

  $(c_1,p_1,m_1) \triangleright (c_2,p_2,m_2) \Rightarrow (c_1 \cdot c, p_1 \cdot p, m_1) \triangleright (c_2 \cdot c, p_2 \cdot p, m_2)$

- Composability of the code :
  if $(c_1 \cdot c_2, p, m) \triangleright^k (\epsilon, p_2, m_2)$ then there exists $k' \in \mathbb{N}$ and a configuration (ϵ, p', m') such that $(c_1, p, m) \triangleright^{k'} (\epsilon, p', m')$ and $(c_2, p', m') \triangleright^{k-k'} (\epsilon, p_2, m_2)$.

The semantic function associated to AM

$$\mathcal{M} : \textbf{Code} \longrightarrow (\textbf{State} \longrightarrow \textbf{State}).$$

$$\mathcal{M}[c]m = \begin{cases} m' & ,(c, \epsilon, m) \rhd^* (\epsilon, p, m') \\ \text{undef, sinon} \end{cases}$$

Code Generation

We need to define three fonctions :

1. $\mathcal{CA}$: **Aexp** $\longrightarrow$ **Code**
2. $\mathcal{CB}$: **Bexp** $\longrightarrow$ **Code**
3. $\mathcal{CS}$: **Stm** $\longrightarrow$ **Code**

such that for any program $c \in$ **Stm**, we have

$$\mathcal{S}_{ns}[\,] = \mathcal{M} \circ \mathcal{CS}$$

We require that $\mathcal{CA}$, $\mathcal{CB}$ and $\mathcal{CS}$ have the following properties:

1. $(\mathcal{CA}[a], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$
2. $(\mathcal{CB}[b], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[b]\sigma, \sigma)$
3. $(\mathcal{CS}[S], \epsilon, \sigma) \triangleright^* (\epsilon, p, \sigma')$ ssi $(S, \sigma) \rightarrow \sigma'$

Code Generation

Some examples for defining $\mathcal{CA}$, $\mathcal{CB}$ et $\mathcal{CS}$:

- $\mathcal{CA}[n] =$ push-n.
- $\mathcal{CA}[x] =$ fetch(x)
- $\mathcal{CA}[a_1 + a_2] = \mathcal{CA}[a_2] \cdot \mathcal{CA}[a_1] \cdot$ add
- $\mathcal{CB}[\text{true}] =$ True
- $\mathcal{CS}[x := a] = \mathcal{CA}[a] \cdot$ store(x)
- $\mathcal{CS}[S_1; S_2] = \mathcal{CS}[S_1] \cdot \mathcal{CS}[S_2]$

Exemple

Axiomatic Semantics: Hoare Logic

Partial and Total Program Correctness

The aim is to specify the behavior of a program as an input/output relation.

Example :

Fact:

$y := 1;$

while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$ od

Partial correctness

If the initial value of x is $n > 0$ and if the program terminates then the final value of y has to be $n!$.

Total correctness

If the initial value of x is $n > 0$ then the program must terminate

and the final value of y has to be $n!$.

Semantic verification

Fact:

$y := 1;$

while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$ od

$(\text{Fact}, \sigma) \rightarrow \sigma'$

$\Downarrow$

$\sigma'(y) = \sigma(x)!$ et $\sigma(x) > 0$

Three steps

1. Correctness of the body of the loop
2. Correction of the loop
3. Correction of the program

Each time, we have to investigate the semantic derivation tree.

Hoare's triple

Example :

$$\{x = n \wedge n > 0\}$$
$$y := 1;$$
while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$ od
$$\{y = n! \wedge n > 0\}$$

- Precondition: $\{x = n \wedge n > 0\}$
- Postcondition: $\{y = n! \wedge n > 0\}$.

We will use first-order logic to express pre- and post-conditions.

Total and partial correctness

The Hoare triple

$$\{P\}S\{Q\}$$

i svalied, denoted by $\models \{P\}S\{Q\}$*, if for every states* σ, σ'*:*

- *if* $\sigma \models P$ *and* $(S, \sigma) \to \sigma'$ *then*
- $\sigma' \models Q$.

We say S is *partially correct* with respect to P and Q.

The Hoare triple

$$[P]S[Q]$$

is valied, denoted by $\models [P]S[Q]$*, if for every* σ*:*

- *if* $\sigma \models P$ *then the program terminates and*
- *for every state* σ'*: if* $(S, \sigma) \to \sigma'$ *then* $\sigma' \models Q$.

We say S is *totally correct* eith respect to P and Q.

Hoare calculus

Axioms:

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

$$\{P\}\text{skip}\{P\}$$

Inference rules

Composition:

$$\frac{\{P\}S_1\{Q\}\ \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

Conditional:

$$\frac{\{b \wedge P\}S_1\{Q\} \quad \{\neg b \wedge P\}S_2\{Q\}}{\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2\{Q\}}$$

Hoare Logic for partial correctness

While:

$$\frac{\{b \wedge P\}S\{P\}}{\{P\}\text{while } b \text{ do } S \text{ od } \{\neg b \wedge P\}}$$

Consequence:

$$\frac{\{P'\}S\{Q'\}}{\{P\}S\{Q\}} \text{ Si } \models P \Rightarrow P' \text{ et } \models Q' \Rightarrow Q$$

Notation:
We write

$$\vdash \{P\}S\{Q\}$$

when $\{P\}S\{Q\}$ is derivable from the axioms and inference rules.

An example

We can derive

- $\{x = n \wedge n > 0\} y := 1 \{x = n \wedge n > 0 \wedge y = 1\}$
- $\{x = n \wedge n > 0 \wedge y = 1\}$
  while $\neg(x = 1)$ do $y := y * x; x := x - 1$ od
  $\{y = n! \wedge n > 0\}$.

and conclude
$\{x = n \wedge n > 0\}$
$y := 1;$ while $\neg(x = 1)$ do $y := y * x; x := x - 1$ od
$\{y = n! \wedge n > 0\}$

The greates common divisor example

Let pgcd denote the following program:
Example :

while $\neg(x = y)$ do $($if $x > y$ then $x := x - y$ else $y := y - x)$ od ;
$z := x$

We can show

$$\{x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0\}\text{pgcd}\{z = \text{pgcd}(n, m)\}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge$ pgcd$(x, y) =$ pgcd(m, n).

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge$ pgcd$(x, y) =$ pgcd(m, n).

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{pre\}\text{pgcd}\{z = pgcd(n, m)\}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

$I: \ x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

$pre: \ x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{pre \Rightarrow I \quad \{I\}\text{pgcd}\{z = \text{pgcd}(n, m)\}}{\{pre\}\text{pgcd}\{z = pgcd(n, m)\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{pre \Rightarrow I \quad \{I\}\text{pgcd}\{z = \text{pgcd}(n, m)\}}{\{pre\}\text{pgcd}\{z = pgcd(n, m)\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{I\}\text{pgcd}\{z = \text{pgcd}(n, m)\}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I\}S'\{x = \text{pgcd}(n, m)\} \quad \{x = \text{pgcd}(n, m)\}z := x\{z = \text{pgcd}(n, m)\}}{\{I\}S'; z := x\{z = \text{pgcd}(n, m)\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I\}S'\{x = \text{pgcd}(n, m)\} \quad \{x = \text{pgcd}(n, m)\}z := x\{z = \text{pgcd}(n, m)\}}{\{I\}S'; z := x\{z = \text{pgcd}(n, m)\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{I\}S'\{x = \text{pgcd}(n, m)\}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I \wedge x \neq y\}S''\{I\} \quad I \wedge x = y \Rightarrow x = \text{pgcd}(n, m)}{\{I\}S'\{x = \text{pgcd}(n, m)\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$
S' : while $\neg(x = y)$ do S'' od
I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.
pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I \wedge x \neq y\} S'' \{I\} \quad I \wedge x = y \Rightarrow x = \text{pgcd}(n, m)}{\{I\} S' \{x = \text{pgcd}(n, m)\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge$ pgcd$(x, y) =$ pgcd(m, n).

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$\{I \wedge x \neq y\}$if $x > y$ then $x := x - y$ else $y := y - x\{I\}$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge$ pgcd$(x, y) =$ pgcd(m, n).

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{\{I \wedge x \neq y \wedge x > y\}x := x - y\{I\} \quad \{I \wedge x \neq y \wedge \neg(x > y)\}y := y - x\{I\}}{\{I \wedge x \neq y\}\text{if } x > y \text{ then } x := x - y \text{ else } y := y - x\{I\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

$I: \; x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

$pre: \; x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge x > y) \Rightarrow I[x - y/x]}{\{I \wedge x \neq y \wedge x > y\} x := x - y \{I\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge x > y) \Rightarrow I[x - y/x]}{\{I \wedge x \neq y \wedge x > y\} x := x - y \{I\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge$ pgcd$(x, y) =$ pgcd(m, n).

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge \neg(x > y)) \Rightarrow I[y - x/x]}{\{I \wedge x \neq y \wedge \neg(x > y)\} y := y - x \{I\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge$ pgcd$(x, y) =$ pgcd(m, n).

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\frac{(I \wedge x \neq y \wedge \neg(x > y)) \Rightarrow I[y - x/x]}{\{I \wedge x \neq y \wedge \neg(x > y)\}y := y - x\{I\}}$$

Proof in Hoare logic

Let S'' : if $x > y$ then $x := x - y$ else $y := y - x$

S' : while $\neg(x = y)$ do S'' od

I : $x \geq 0 \wedge y \geq 0 \wedge \text{pgcd}(x, y) = \text{pgcd}(m, n)$.

pre : $x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0$.

$$\{pre\}\text{pgcd}\{z = pgcd(n, m)\}$$

Soundness and Completeness

Soundness:

if $\vdash \{P\}S\{Q\}$ then $\models \{P\}S\{Q\}$, i.e.,

we can derive only valid statements.

Completeness

if $\models \{P\}S\{Q\}$ then $\vdash \{P\}S\{Q\}$, i.e.,

all valid statements are derivable.

Soundness of Hoare Logic

Théorème Hoare logic for partial correctness is sound

$$\text{If } \vdash \{P\}S\{Q\} \text{ then } \models \{P\}S\{Q\}$$

The proof is by induction on the derivation tree.

Completeness of Hoare Logic

Théorème Hoare logic for partial correctness is complete

If $\models \{P\}S\{Q\}$ then $\vdash \{P\}S\{Q\}$

The plan of the proof

We introduce $\mathsf{wlp}(S, Q)$ as follows :

- $\models \{P\}S\{Q\}$ iff $\models P \Rightarrow \mathsf{wlp}(S, Q)$ and
- $\vdash \{\mathsf{wlp}(S, Q)\}S\{Q\}$

$\mathsf{wlp}(S, Q)$ describes the following set:

$$\{\sigma \mid \forall \sigma' \cdot (S, \sigma) \rightarrow \sigma' \Rightarrow \sigma' \models Q\}$$

For now, we assume that we can express $\mathsf{wlp}(S, Q)$ in first-order logic.
The proof of the first item is immediate.
The second item can be proved by induction of the structure of programs.

Case of the While-statement

$$S \equiv \text{while } b \text{ do } S' \text{ od}$$

To prove

$$\vdash \{\text{wlp}(S,Q)\}S\{Q\}$$

it is enough to prove:

1. $\models \neg b \land \text{wlp}(S,Q) \Rightarrow Q$
2. $\models b \land \text{wlp}(S,Q) \Rightarrow \text{wlp}(S', \text{wlp}(S,Q))$.

since

$$\cfrac{\cfrac{\cfrac{2. \quad \{\text{wlp}(S',\text{wlp}(S,Q))\}S'\{\text{wlp}(S,Q)\} \; (I.H.)}{\{b \land \text{wlp}(S,Q)\}S'\{\text{wlp}(S,Q)\}}}{\{\text{wlp}(S,Q)\}S\{\neg b \land \text{wlp}(S,Q)\}} \qquad 1.}{\{\text{wlp}(S,Q)\}S\{Q\}}$$

Expressiveness and Decidability

We made the assumption that $\mathsf{wlp}(S, Q)$ is expressible in 1st-order logic. Is this assumption reasonable?
It is, if our 1st-order logic includes Peano arithmetic $(\mathbb{N}, +, *)$.
Reason: this allows to encode sequences of integers as integers.
This is called Gëdelisation.
Consequence: Hoare logic is undecidable $\{P\}S\{Q\}$.
The complement of

$$\{\{P\}S\{\mathsf{false}\} \mid\ \models \{P\}S\{Q\}\}$$

is recursively enumerable but not recursive.

Termination Proofs

Recall that

$$[P]S[Q]$$

is valied, denoted by $\models [P]S[Q]$, if for every state σ:

- *if $\sigma \models P$ then the program terminates and*
- *for evry state σ': if $(S, \sigma) \to \sigma'$ then $\sigma' \models Q$.*

Notice that $\models [P]S[Q]$ iff $\models \{P\}S\{Q\}$ and $\models [P]S[true]$
In other words,
Total correctness = Partical correction $\wedge$ termination.

Well-founded ordering

Let $\leq$ be a relation on A.
$\leq$ is an ordering on A, if

1. $\leq$ is reflexive: $\forall a \in A \cdot a \leq a$.
2. $\leq$ is anti-symmetric: $\forall a, b \in A \cdot a \leq b \wedge b \leq a \Rightarrow a = b$.
3. $\leq$ is transitive: $\forall a, b, c \in A \cdot a \leq b \wedge b \leq c \Rightarrow a \leq c$.

A total (also called linear) ordering, is an ordering such that a $a \leq b$ or $b \leq a$, for every $a, b \in A$.
Let $a < b$ iff $a \leq b \wedge a \neq b$.

An ordering $\leq$ is *well-founded* if it does not include an infinitely decreasing chain $a_0 > a_1 > a_2 \cdots$. A *well-order* est well-founded linear ordering.

Examples

- $(\mathbb{N}, \leq)$ is well-order.
- $(\Sigma^*, \preceq)$ is a well-founded ordering.
- $(2^A, \subseteq)$ is a well-founded ordering.
- Let $(P, \leq)$ be an ordering. We define$\leq_L$ on $P^* \times P^*$ as follows:
  $u \leq_L v$ iff
  - u is prefixe of v or
  - there $i < \min(|u|, |v|)$ such that $u(i) < v(i)$ and $\forall j \in [0, i) \cdot u(j) = v(j)$.

  if $(P, \leq)$ is an ordering then $(P^*, \leq_L)$ is an ordering.

Examples

- Let $(P, \leq)$ be an ordering. We define $\leq_{lex}^n$ on $P^n \times P^n$ as follows:
  $u \leq_{lex}^n v$ iff
  - u is prefix of v or
  - there is $i < n$ such that $u(i) < v(i)$ and $\forall j \in [0, i) \cdot u(j) = v(j)$.

  If $(P, \leq)$ is well-founded ordering then $(P^n, \leq_{lex}^n)$ is also well-founded ordering.
- We extend $\leq_{lex}^n$ to words of any length $u \leq_{lex} v$ iff
  - u is prefixe of v or
  - $|u| = |v|$ and $u \leq_{lex}^{|u|} v$.

  If $(P, \leq)$ is a well-founded ordering then $(P^*, \leq_{lex})$ is also a well-founded ordering.
- $(\mathbb{N}^*, \leq_{lex})$ is a well-founded ordering.

Loop Termination

Théorème

$$[P]\text{while } b \text{ do } S' \text{ od } [true] \text{ est vrai}$$

iff there is a well-founded ordering $(W, \leq)$, a function f : **State** $\rightarrow W$ (ranking) and an assertion I such that

1. $\models b \wedge P \Rightarrow I$ and $\models \{I \wedge B\}S'\{I\}$, i.e., I is an invariant of the loop and
2. $\models [b \wedge I \wedge f = z]S'[f < z]$, i.e., the ranking function decreases at each iteration.

Example

while $\neg(x = y)$ do (if $x > y$ then $x := x - y$ else $y := y - x$) od ;
$z := x$

We showed

$$\{x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0\}\text{pgcd}\{z = \text{pgcd}(n, m)\}.$$

And termination?

Denotational Semantics

Motivation

- The operational semantics describes how a program is executed.
- The axiomatic semantics allows us to reason about programs while abstracting the details of its execution.
- The denotational semantics describes the meaning (denotation) of a programs without describes how it is executed.
  Denotational semantics is compositional.

Semantic clauses

$$\mathcal{S}_d : \textbf{Stm} \longrightarrow \left(\textbf{State} \stackrel{part.}{\longrightarrow} \textbf{State}\right)$$

$$\mathcal{S}_d[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$\mathcal{S}_d[\![\mathsf{skip}]\!] = id$$

$$\mathcal{S}_d[\![S_1; S_2]\!] = \mathcal{S}_d[\![S_2]\!] \circ \mathcal{S}_d[\![S_1]\!]$$

$$\mathcal{S}_d[\![\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!] = \mathsf{cond}(\mathcal{B}[b], \mathcal{S}_d[\![S_1]\!], \mathcal{S}_d[\![S_2]\!])\ \text{where}$$

$$\mathsf{cond}(p, f, g)\sigma = \begin{cases} f(\sigma);\ \text{if } p(\sigma) = \textbf{tt} \\ g(\sigma);\ \text{otherwise} \end{cases}$$

Denotation of Loops

We should have

$$\mathcal{S}_d[\![\text{while } b \text{ do } S \text{ od }]\!] = \text{cond}(\mathcal{B}[B], \mathcal{S}_d[\![\text{while } b \text{ do } S \text{ od }]\!] \circ \mathcal{S}_d[\![S]\!], id)$$

Hence, $\mathcal{S}_d[\![\text{while } b \text{ do } S \text{ od }]\!]$ is a solution of the equation:

$$f = \text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id)$$

Let $F(f) = \text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id)$.
Does this equation has solutions? several?
We shall see that this equation has always solutions and that the least one is the one we are interested in:
For every f such that $f = F(f)$, we have:

$$\mathcal{S}_d[\![\text{while } b \text{ do } S \text{ od }]\!]\sigma = \sigma' \Rightarrow f(\sigma) = \sigma'.$$

Fixpoints

when does $F(X) = X$ has a solution?
To answer this question we rely on

fixpoint theory

Upper bound, lower bound and limits

Let $(D, \sqsubseteq)$ be an ordered set and $X \subseteq D$.

- d is a *lower bound* of X, if $d \sqsubseteq x$, for every $x \in X$.
- d is a *upper bound* of X, if $x \sqsubseteq d$, for every $x \in X$.
- d is a *greatest lower bound (g.l.b.)* of X, denoted by $\sqcap X$, if d is a lower bound X and for every lower bound y: $y \sqsubseteq x$.
- d is *least upper bound (l.u.b.)* of X, denoted by $\sqcup X$, if d is an upper bound of X and for every upper bound y: $x \sqsubseteq y$.

Let $\bot$ denote $\sqcap D$ and $\top$ denote $\sqcup D$, when they exist.
Example:

Let $(2^A, \subseteq)$. Then, $\sqcap X = \bigcap_{x \in X} x$ and $\sqcup X = \bigcup_{x \in X} x$.

Lattice

An ordered set $(D, \sqsubseteq)$ is a *lattice*, if $\sqcap X$ and $\sqcup X$ exist for every finite $X \subseteq D$.
It is a *complete lattice*, if $\sqcap X$ and $\sqcup X$ exist for every $X \subseteq D$.
Examples:

1. $(2^A, \subseteq)$ is a complete lattice.
2. $(\{X \mid X \subseteq A, X \text{ finite}\}, \subseteq)$ is lattice that is not complte, when A is infinite.
3. Let Σ be a finite alphabet. Then, $(\Sigma^*, \preceq)$ is lattice which is not complete

Continuous functions

Let $(D, \sqsubseteq)$ be an ordered set. A *chain* (ω-chain) is an enumerable subset $X \subseteq D$ such that $\sqsubseteq$ is total on X. I.e., the elements of X can be ordered $x_0 \sqsubseteq x_1 \cdots$.
Definition: Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ordered sets.
A function $f : D \to D'$ is continuous, if

1. it is monotonic: $d_1 \sqsubseteq d_2$ implies $f(d_1) \sqsubseteq' f(d_2)$ and
2. it is $\sqcup$-additive: $f(\sqcup X) = \sqcup' f(X)$, for every chain X.

The Theorems of Knaster-Tarski and Kleene

Théorème Let $(D, \sqsubseteq)$ be a complete lattice and $f : D \rightarrow D$.

- Knaster-Tarski. If f is monotonic then $\sqcap\{x \mid f(x) \sqsubseteq x\}$ is the least fixpoint of f.
- Kleene. If f is continuous then $\bigsqcup_{i \geq 0} f^i(\bot)$ is the least fixpoint of f, where $\bigsqcup_{i \geq 0} f^i(\bot) = \bigsqcup\{f^i(\bot) \mid i \geq 0\}$.

Proof of Knaster-Tarski's Theorem

Let $(D, \sqsubseteq)$ be a complete lattice and $f : D \to D$ be a monotonic function. Let $A = \{x \mid f(x) \sqsubseteq x\}$ and $x_0 = \sqcap A$.

1. x_0 is a fixpoint
   (a) we show that $x \in A$ implies $f(x) \in A$. Let $x \in A$. Then, $f(x) \sqsubseteq x$. Since f is monotonic, $f(f(x)) \sqsubseteq f(x)$. Hence, $f(x) \in A$.
   (b) We show $x_0 \in A$. By definition of x_0, for every $x \in A$ we have $x_0 \sqsubseteq x$. Hence,since f is monotonic, for every $x \in A$ we have $f(x_0) \sqsubseteq f(x) \sqsubseteq x$. Therefore, $f(x_0)$ is a lower bound of A. Since x_0 is the greatest lower bound of A, we have $f(x_0) \sqsubseteq x_0$. That is, $x_0 \in A$.

   (a) and (b) together imply $f(x_0) \in A$. Hence, $x_0 \sqsubseteq f(x_0)$ and $f(x_0) = x_0$.
2. x_0 is the least fixpoint. For every fixpoint of f is in A et x_0 is a lower bound of A.

Proof of Kleene's Theorem

Let $(D, \sqsubseteq)$ be a complete lattice and $f : D \to D$ continuous.

1. $\bigsqcup_{i \geq 0} f^i(\bot)$ is a fixpoint

$$f(\bigsqcup_{i \geq 0} f^i(\bot)) = \bigsqcup_{i \geq 0} f^{i+1}(\bot) = \bigsqcup_{i \geq 1} f^i(\bot) = \bot \sqcup \bigsqcup_{i \geq 1} f^i = \bigsqcup_{i \geq 0} f^i(\bot)$$

2. $\bigsqcup_{i \geq 0} f^i(\bot)$ is lower than any fixpoint f.
   Let x be a fixpoint. We show by induction on i:
   $\forall i \geq 0 \cdot f^i(\bot) \sqsubseteq x$. This implies $\bigsqcup_{i \geq 0} f^i(\bot) \sqsubseteq x$.

Ordre sur les états

Let $(\textbf{State} \stackrel{part.}{\longrightarrow} \textbf{State}, \sqsubseteq)$ be defined by: $f \sqsubseteq g$ iff for every σ, if $f(\sigma)$ is defined then $g(\sigma)$ is also defined and $f(\sigma) = g(\sigma)$.
Then, $\sqcap X$ exists for every X but not $\sqcup X$.
For this reason, let

$$\textbf{State}_{\bot}^{\top} = \textbf{State} \cup \{\top\} \cup \{\bot\}$$

equipped with the following order: $\sigma \sqsubseteq \sigma'$ iff

1. $\sigma' = \top$ or
2. $\sigma = \bot$ or
3. $\sigma = \sigma'$

We can show that

$$(\textbf{State}_{\bot}^{\top}, \sqsubseteq) \text{ is a complete lattice.}$$

Ordering Functions

Let $\mathcal{D}$ be the set of total continuous functions from **State**$_{\perp}^{\top}$ to **State**$_{\perp}^{\top}$.

We define the following ordering $\mathcal{D}$:

$f \sqsubseteq g$ iff $f(\sigma) \sqsubseteq g(\sigma)$, for every σ.

Théorème $(\mathcal{D}, \sqsubseteq)$ is a complete lattice.

Denotation of Loops

Notation: Let $f : A \rightarrow A$. We denote by μf, resp. νf, the least, resp. greatest, fixpoint of f, if it exists.
We define

$$\mathcal{S}_d[\![\text{while } b \text{ do } S \text{ od }]\!] = \mu(\lambda f.\text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id)).$$

Fixpoint theory tells us that $\mathcal{S}_d[\![\text{while } b \text{ do } S \text{ od }]\!]$ exists, if $\lambda f.\text{cond}(\mathcal{B}[B], f \circ \mathcal{S}_d[\![S]\!], id))$ is monotonic.

We can show that this functional is even continuous.