Semantics of Programming Languages

—

How to Prove it

Tobias Nipkow

Fakultät für Informatik
TU München

Wintersemester 2010

1 Introduction

Why Semantics?

Without semantics,
we do not really know what our programs mean.

We merely have a good intuition and a warm feeling.

Like the state of mathematics in the 19th century
— before set theory and logic entered the scene.

Intuition is important!

- You need a good intuition to get your work done efficiently.
- To understand the average accounting program, intuition suffices.
- To write a bug-free accounting program may require more than intuition!
- I assume you have the necessary intuition.
- This course is about “beyond intuition”.

Intuition is not sufficient!

Writing correct language processors (e.g. compilers, refactoring tools, . . .) requires

- a deep understanding of language semantics,
- the ability to *reason* (= perform proofs) about the language and your processor.

Example:
What does the correctness of a type checker even mean? How is it proved?

Why Semantics??

We have a compiler — that is the ultimate semantics!!

- A compiler gives each individual program a semantics.
- It does not help with reasoning about the PL or individual programs.
- Because compilers are far too complicated.
- They provide the worst possible semantics.
- Moreover: compilers may differ!

The sad facts of life

- Most languages have one or more compilers.
- Most compilers have bugs.
- Few languages have a (separate, abstract) semantics.
- If they do, it will be informal (English).

Bugs

- Google “compiler bug”
- Google “hostile applet”
  Early versions of Java had various security holes. Some of them had to do with an incorrect *bytecode verifier*.

  GI Dissertationspreis 2003:
  Gerwin Klein: *Verified Java Bytecode Verification*

Standard ML (SML)

First real language with a mathematical semantics:
Milner, Tofte, Harper:
The Definition of Standard ML. 1990.

Robin Milner (1934–2010)
Turing Award 1991.

Main achievements: LCF (theorem proving)
SML (functional programming)
CCS, pi (concurrency)

The sad fact of life

SML semantics hardly used:

- too difficult to read to answer simple questions quickly
- too much detail to allow reliable informal proof
- not processable beyond LaTeX, not even executable

More sad facts of life

- Real programming languages *are* complex.
- Even if designed by academics, not industry.
- Complex designs are error-prone.
- Informal mathematical proofs of complex designs are also error-prone.

The solution

Machine-checked language semantics and proofs

- Semantics at least type-correct
- Maybe executable
- *Proofs machine-checked*

The tool:

Interactive Theorem Prover
(ITP)

Interactive Theorem Provers

- You give the structure of the proof
- The ITP checks the correctness of each step
- Can prove hard and huge theorems

Government health warnings:

Time consuming
Potentially addictive
Undermines your naive trust in informal proofs

Terminology

This lecture course:

Formal = machine-checked
Verification = formal correctness proof

Traditionally:

Formal = mathematical

Two landmark verifications

C compiler
Competitive with `gcc -O1`

Xavier Leroy
INRIA Paris
using Coq

Operating system
microkernel (L4)

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

A happy fact of life

Programming language researchers are increasingly using ITPs

Why verification pays off

Short term: *The software works!*

Long term:

Tracking effects of changes by rerunning proofs

Incremental changes of the software
typically require only incremental changes of the proofs

Long term much more important than short term:

Software Never Dies

1. Introduction
   - Background
   - This Course

What this course is *not* about

- Hot or trendy PLs
- Comparison of PLs or PL paradigms
- Compilers (although they will be one application)

What this course *is* about

- Techniques for the description and analysis of
  - PLs
  - PL tools
  - Programs
- Description techniques: *operational semantics*
- Proof techniques: *inductions*

Both informally and formally (ITP!)

Our ITP: Isabelle/HOL

- Developed mainly in Munich (Nipkow & Co) and Paris (Wenzel)
- Started 1986 in Cambridge (Paulson)
- The logic HOL is ordinary mathematics

Learning to use Isabelle/HOL
is an integral part of the course

All exercises require the use of Isabelle/HOL

Why I am so passionate about the ITP part

- It is the future
- It is the only way to deal with complex languages *reliably*
- I want students to learn how to write correct proofs
- I have seen too many proofs that look more like LSD trips than coherent mathematical arguments

Overview of course

- Introduction to Isabelle/HOL
- IMP (assignment and while loops) and its semantics
- A compiler for IMP
- Hoare logic for IMP
- Type systems for IMP
- Program analysis for IMP

The semantics part of the course is mostly traditional

The use of an ITP is leading edge

So far, there are only a handful of universties that combine these two topics as aggressively as we do: Harvard, Princeton, UPenn, Saarbrücken, . . .

What you learn in this course goes far beyond PLs

It has applications in compilers, security, software engineering etc.

It is a new approach to informatics

Part I

Programming and Proving in HOL

2. Overview of Isabelle/HOL

3. Type and function definitions

4. Simplification and Induction

Notation

Implication associates to the right:

$$A \Longrightarrow B \Longrightarrow C \quad \text{means} \quad A \Longrightarrow (B \Longrightarrow C)$$

$$\frac{A_1 \quad \ldots \quad A_n}{B} \quad \text{means} \quad A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

2. Overview of Isabelle/HOL

3. Type and function definitions

4. Simplification and Induction

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:

- For the moment: only $term = term$, e.g. $1 + 2 = 4$
- Later: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, ...

2 Overview of Isabelle/HOL

- Types and terms
- Proof General
- By example: types *bool*, *nat* and *list*
- Summary

Types

Basic type syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	$\tau\ list$	lists
	$\mid$	$\tau\ set$	sets
	$\mid$	$\ldots$	user-defined types

Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

Terms

Terms can be formed as follows:

- Function application:
  $f\ t$
  is the call of function f with argument t.
  If f has more arguments: $f\ t_1\ t_2\ \ldots$
  Examples: $sin\ \pi$, $plus\ x\ y$
- Function abstraction:
  $\lambda x.\ t$
  is the function with parameter x and result t,
  i.e. "$x \mapsto t$".
  Example: $\lambda x.\ plus\ x\ x$

Basic term syntax:

$$
\begin{array}{rcll}
t & ::= & (t) & \\
& | & a & \text{constant or variable (identifier)} \\
& | & t\ t & \text{function application} \\
& | & \lambda x.\ t & \text{function abstraction} \\
& | & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

Examples: $f\ (g\ x)\ y$
$h\ (\lambda x.\ f\ (g\ x))$

Convention: $f\ t_1\ t_2\ t_3 \ \equiv\ ((f\ t_1)\ t_2)\ t_3$

This language of terms is known as the λ-calculus.

The computation rule of the λ-calculus is the replacement of formal by actual parameters:

$$(\lambda x.\ t)\ u\ =\ t[u/x]$$

where $t[u/x]$ is "t with u substituted for x".

Example: $(\lambda x.\ x + 5)\ 3\ =\ 3 + 5$

- The step from $(\lambda x.\ t)\ u$ to $t[u/x]$ is called β-reduction.
- Isabelle performs β-reduction automatically.

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:

$t :: \tau$ means "t is a well-typed term of type τ".

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\ u :: \tau_2}$$

Type inference

Isabelle automatically computes the type of each variable in a term. This is called type inference.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

User can help with type annotations inside the term.
Example: $f\ (x{::}nat)$

Currying

Thou shalt Curry your functions

- Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Advantage:

Currying allows *partial application*

$f\ a_1$ where $a_1 :: \tau_1$

Predefined syntactic sugar

- *Infix:* $+$, $-$, $*$, $\#$, $@$, ...
- *Mixfix:* if _ then _ else _, case _ of, ...

Prefix binds more strongly than infix:

! $f\ x + y \ \equiv\ (f\ x) + y \ \not\equiv\ f\ (x + y)$ **!**

Enclose *if* and *case* in parentheses:

! $(\mathit{if}$ _ then _ else _$)$ **!**

Isabelle text = Theory = Module

Syntax:

```
theory MyTh
imports ImpTh_1 ... ImpTh_n
begin
(definitions, theorems, proofs, ...)*
end
```

$MyTh$: name of theory. Must live in file $MyTh$`.thy`

$ImpTh_i$: name of *imported* theories. Import transitive.

Usually: `imports Main`

Proof General

An Isabelle Interface

by David Aspinall

Proof General

Customized version of (x)emacs:

- all of emacs
- Isabelle aware (when editing `.thy` files)
- mathematical symbols ("x-symbols")

X-Symbols

Input of funny symbols

- via abbreviation: =>, ==>, /\, \/, . . .
- via ascii encoding (similar to LaTeX): \<and>, . . .
- via menu ("X-Symbol")

I3P by Holger Gast

Similar to ProofGeneral but

- Does not need emacs
- $\Longrightarrow$ easier to install!
- Based on Netbeans/Swing
- $\Longrightarrow$ may be more familiar
- Nicer fonts
- ...

Concrete syntax

In `.thy` files:

Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides

Overview_Demo.thy

2 Overview of Isabelle/HOL

Type $bool$

datatype $bool \;=\; True \;|\; False$

Predefined functions:

$\wedge,\ \vee,\ \longrightarrow,\ \ldots\ ::\ bool \Rightarrow bool \Rightarrow bool$

A logical formula is a term of type $bool$

if-and-only-if: $=$

Type nat

datatype $nat = 0 \mid Suc\ nat$

Values of type nat: 0, $Suc\ 0$, $Suc(Suc\ 0)$, ...

Predefined functions: $+, *, ... :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,... :: 'a, \quad + :: 'a \Rightarrow 'a \Rightarrow 'a$

You need type annotations: $1 :: nat$, $x + (y::nat)$
... unless the context is unambiguous: $Suc\ z$

`Nat_Demo.thy`

Type $'a\ list$

Lists of elements of type $'a$

datatype $'a\ list\ =\ Nil\ |\ Cons\ 'a\ ('a\ list)$

Syntactic sugar:

- $[] = Nil$: empty list
- $x \# xs = Cons\ x\ xs$:
  list with first element x ("*head*") and rest xs ("*tail*")
- $[x_1, \ldots, x_n] = x_1 \# \ldots\ x_n \# []$

Structural Induction for lists

To prove that $P(xs)$ for all lists xs, prove

- $P([])$ and
- for arbitrary x and xs, $P(xs)$ implies $P(x\#xs)$.

$$\frac{P([]) \qquad \bigwedge x\ xs.\ P(xs) \Longrightarrow P(x\#xs)}{P(xs)}$$

`List_Demo.thy`

Large library: `HOL/List.thy`

Included in `Main`.

Don't reinvent, reuse!

Predefined: $xs\ @\ ys$ (append), $length$, and map:

$$map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

fun $map :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
$map\ f\ [] = []\ \ |$
$map\ f\ (x\#xs) = f\ x\ \#\ map\ f\ xs$

Note: map takes *function* as argument.

2 Overview of Isabelle/HOL

- **datatype** defines (possibly) recursive data types.
- **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.

Proof methods

- *induct* performs structural induction on some variable (if the type of the variable is a datatype).

- *auto* solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

  "$=$" is used only from left to right!

Proofs

General schema:

lemma *name*: `"..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name* `[simp]: "..."`

Top down proofs

Command

sorry

"completes" any proof.

Allows top down development:

Assume lemma first, prove it later.

The proof state

$1.\ \bigwedge x_1 \ \ldots \ x_p.\ \ A \Longrightarrow B$

$x_1 \ \ldots \ x_p$	fixed local variables
A	local assumption(s)
B	actual (sub)goal

Preview: Multiple assumptions

$$[\![\ A_1; \ \dots\ ; A_n\ ]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

$;\quad \approx\quad$ "and"

2 Overview of Isabelle/HOL

3 Type and function definitions

4 Simplification and Induction

3 Type and function definitions

Type definitions

Function definitions

Type abbreviations

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types

$$name = string$$
$$('a,'b)foo = 'a\ list \times 'b\ list$$

Type abbreviations are expanded after parsing
and are not present in internal representation and output

datatype — the general case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \;=\; C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \;\mid\; \ldots \;\mid\; C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:* $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \cdots \wedge x_{n_i} = y_{n_i})$

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Case expressions

Datatype values can be taken apart with $case$:

$$(case\ xs\ of\ \ [] \Rightarrow \ldots \ \ |\ \ y\#ys \Rightarrow \ldots\ y\ \ldots\ ys\ \ldots)$$

Wildcards: _

$$(case\ m\ of\ \ 0 \Rightarrow Suc\ 0\ \ |\ \ Suc\ \_ \Rightarrow 0)$$

Nested patterns:

$$(case\ xs\ of\ \ [0] \Rightarrow 0\ \ |\ \ [Suc\ n] \Rightarrow n\ \ |\ \ \_ \Rightarrow 2)$$

Complicated patterns mean complicated proofs!

Need ( ) in context

3 Type and function definitions

Type definitions

Function definitions

Non-recursive definitions

Example:

definition $sq :: nat \Rightarrow nat$ **where** $sq\ n\ =\ n{*}n$

No pattern matching, just $f\, x_1\ \ldots\ x_n\ =\ \ldots$

Nontermination can kill

How about $f\,x = f\,x + 1$?

! All functions in HOL must be total **!**

Key features of **fun**

- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
- Proves customized induction schema

Example: separation

fun $sep :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

$sep\ a\ (x \# y \# zs) = x\ \#\ a\ \#\ sep\ a\ (y \# zs)\ \ |$
$sep\ a\ xs = xs$

Example: Ackermann

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**
$ack\ 0\ n = Suc\ n\ |$
$ack\ (Suc\ m)\ 0 = ack\ m\ (Suc\ 0)\ |$
$ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)$

Terminates because the arguments decrease *lexicographically* with each recursive call:

- $(Suc\ m,\ 0) > (m,\ Suc\ 0)$
- $(Suc\ m,\ Suc\ n) > (Suc\ m,\ n)$
- $(Suc\ m,\ Suc\ n) > (m,\ \_)$

Tree_Demo.thy

primrec

- A restrictive version of **fun**
- Means *primitive rercursive*
- Most functions are primitive recursive
- Frequently found in Isabelle theories

The essence of primitive recursion:

$$\begin{array}{lll} f(0) & = \dots & \text{no recursion} \\ f(Suc\ n) & = \dots f(n) \dots & \end{array}$$

$$\begin{array}{lll} g([]) & = \dots & \text{no recursion} \\ g(x\#xs) & = \dots g(xs) \dots & \end{array}$$

2 Overview of Isabelle/HOL

3 Type and function definitions

4 Simplification and Induction

4 Simplification and Induction

Simplification

Induction

Simplification means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ *simplification rule*

Simplification = (Term) Rewriting

An example

Equations:

$$
\begin{array}{rcll}
0 + n & = & n & (1) \\
(Suc\ m) + n & = & Suc\ (m + n) & (2) \\
(Suc\ m \leq Suc\ n) & = & (m \leq n) & (3) \\
(0 \leq m) & = & True & (4)
\end{array}
$$

Rewriting:

$$
\begin{array}{rcll}
0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\
Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\
Suc\ 0 & \leq & Suc\ (0 + x) & \stackrel{(3)}{=} \\
0 & \leq & 0 + x & \stackrel{(4)}{=} \\
 & True & &
\end{array}
$$

Conditional rewriting

Simplification rules can be conditional:

$$[\![\ P_1; \ \ldots; \ P_k \ ]\!] \Longrightarrow l = r$$

is applicable only if all P_i can be proved first, again by simplification.

Example:

$$\begin{array}{rcl} p(0) & = & True \\ p(x) \Longrightarrow \ f(x) & = & g(x) \end{array}$$

We can simplify $f(0)$ to $g(0)$ but
we cannot simplify $f(1)$ because $p(1)$ is not provable.

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

$$\llbracket\ P_1; \ldots; P_k\ \rrbracket \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

$$n < m \Longrightarrow (n < Suc\ m) = True \quad \text{YES}$$
$$Suc\ n < m \Longrightarrow (n < m) = True \quad \text{NO}$$

Proof method *simp*

Goal: 1. $\llbracket\ P_1; \ldots; P_m\ \rrbracket \Longrightarrow C$

apply$(simp\ add\colon\ eq_1\ \ldots\ eq_n)$

Simplify $P_1\ \ldots\ P_m$ and C using

- lemmas with attribute $simp$
- rules from **fun** and **datatype**
- additional lemmas $eq_1\ \ldots\ eq_n$
- assumptions $P_1\ \ldots\ P_m$

Variations:

- $(simp\ \ldots\ del\colon\ \ldots)$ removes $simp$-lemmas
- add and del are optional

auto versus *simp*

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1

- *auto* applies *simp* and more

- *auto* can also be modified:
  $(auto\ simp\ add\!:\ \ldots\ simp\ del\!:\ \ldots)$

Rewriting with definitions

Definitions (**definition**) must be used explicitly:

$$(simp\ add{:}\ f\_def \ldots)$$

f is the function whose definition is to be unfolded.

Case splitting with *simp*

Automatic:

$$P(\mathit{if}\ A\ \mathit{then}\ s\ \mathit{else}\ t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

By hand:

$$P(\mathit{case}\ e\ \mathit{of}\ 0 \Rightarrow a \mid \mathit{Suc}\ n \Rightarrow b)$$
$$=$$
$$(e = 0 \longrightarrow P(a)) \wedge (\forall\, n.\ e = \mathit{Suc}\ n \longrightarrow P(b))$$

Proof method: (*simp split: nat.split*)
Or *auto*. Similar for any datatype t: *t.split*

Simp_Demo.thy

4 Simplification and Induction

Simplification

Induction

Basic induction heuristics

Theorems about recursive functions are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

A tail recursive reverse

Our initial reverse:

fun $rev :: 'a\ list \Rightarrow 'a\ list$ **where**
$rev\ [] = []\ \ |$
$rev\ (x\#xs) = rev\ xs\ @\ [x]$

A tail recursive version:

fun $itrev :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$itrev\ []\ \ ys = ys\ \ |$
$itrev\ (x\#xs)\ \ ys =$

lemma $itrev\ xs\ [] = rev\ xs$

Why in this direction?
Because the lhs is "more complex" than the rhs.

Induct_Demo.thy

Generalisation

Generalisation

- Replace constants by variables
- Generalize free variables
  - by $\forall$ in formula
  - by $arbitrary$ in induction proof

So far, all proofs were by structural induction
because all functions where primitive recursive.

In each induction step, 1 constructor is added.
In each recursive call, 1 constructor is removed.

Now: induction for complex recursion patterns.

Computation Induction: Example

fun $div2 :: nat \Rightarrow nat$ **where**
$div2\ 0 = 0\ \ |$
$div2\ (Suc\ 0) = 0\ \ |$
$div2(Suc(Suc\ n)) = Suc(div2\ n)$

$\rightsquigarrow$ induction rule `div2.induct`:

$$\frac{P(0) \quad P(Suc\ 0) \quad P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

for each defining equation

$$f(e) \;=\; \ldots f(r_1) \ldots f(r_k) \ldots$$

prove $P(e)$ *assuming* $P(r_1), \ldots, P(r_k)$.

Induction follows course of (terminating!) computation
Motto: properties of f are best proved by rule $f.induct$

How to apply $f.induct$

If $f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau'$:

$$(induct\ a_1\ \ldots\ a_n\ rule{:}\ f.induct)$$

Heuristic:

- there should be a call $f\ a_1\ \ldots\ a_n$ in your goal
- ideally the a_i should be variables.

Induct_Demo.thy

Computation Induction

Part II

Interlude: Expressions

5 IMP Expressions

5 IMP Expressions

This section introduces

arithmetic and boolean expressions

of our imperative language IMP.

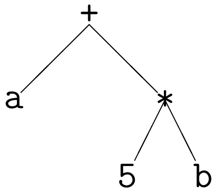
IMP *commands* are introduced later.

5. IMP Expressions
   - Arithmetic Expressions
   - Boolean Expressions
   - Stack Machines and Compilation

Concrete and abstract syntax

Concrete syntax: strings, eg "a+5*b"

Abstract syntax: trees, eg



Parser: function from strings to trees

Linear view of trees: terms, eg $Plus\ a\ (Times\ 5\ b)$

Abstract syntax trees/terms are datatype values!

Concrete syntax is defined by a context-free grammar, eg

$$a ::= n \mid x \mid (a) \mid a + a \mid a * a \mid \ldots$$

where n can be any natural number and x any variable.

We focus on *abstract* syntax
which we introduce via datatypes.

Datatype *aexp*

Variable names are replaced by numbers:

types $name = nat$
datatype $aexp = N\ nat \mid V\ name \mid Plus\ aexp\ aexp$

Concrete	Abstract
5	$N\ 5$
x	$V\ 0$
x+y	$Plus\ (V\ 0)\ (V\ 1)$
2+(z+3)	$Plus\ (N\ 2)\ (Plus\ (V\ 2)\ (N\ 3))$

Warning

This is syntax, not (yet) semantics!

$$N\ 0 \neq Plus\ (N\ 0)\ (N\ 0)$$

The (program) state

What is the value of `x+1`?

- The value of an expression
  depends on the value of its variables.
- The value of all variables is recorded in the *state*.
- The state is a function from variable names to values:

types $state = name \Rightarrow nat$

How to write down a state

- There is no pretty notation like $\{0 \mapsto 7,\ 1 \mapsto 42,\ \ldots\}$
- But there is $[7,\ 42,\ \ldots]$
- And there is $nth :: 'a\ list \Rightarrow (nat \Rightarrow 'a)$
- Thus: $nth\ [7,\ 42,\ \ldots] :: state$
  $nth\ [7,\ 42,\ \ldots] \approx \{0 \mapsto 7,\ 1 \mapsto 42,\ \ldots\}$
  The joys of *partial application*!
- Infix syntax for $nth\ xs\ n$: $xs\ !\ n$
- By def of nth: $[7,\ 42,\ \ldots]\ !\ 1 = 42$
- Warning: $[7,\ 42]\ !\ 3$ has some value — but we do not know which!

AExp.thy

5 IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machines and Compilation

`BExp.thy`

5 IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machines and Compilation

ASM.thy

This was easy.
Because evaluation of expressions always terminates.
But execution of programs may *not* terminate.
Hence we cannot define it by a total recursive function.

We need more logical machinery
to define program execution and reason about it.

Part III

Logic and Structured Proofs

6 Logic and Proof beyond "="

7 Isar: A Language for Structured Proofs

6 Logic and Proof beyond "="

7 Isar: A Language for Structured Proofs

6 Logic and Proof beyond "="

Logical Formulas

Proof Automation

Single Step Proofs

Inductive Definitions

Syntax (in decreasing priority):

$$
\begin{array}{lllllll}
\mathit{form} & ::= & (\mathit{form}) & \mid & \mathit{term} = \mathit{term} & \mid & \neg \mathit{form} \\
 & \mid & \mathit{form} \wedge \mathit{form} & \mid & \mathit{form} \vee \mathit{form} & \mid & \mathit{form} \longrightarrow \mathit{form} \\
 & \mid & \forall x.\ \mathit{form} & \mid & \exists x.\ \mathit{form} & &
\end{array}
$$

Examples:

$$
\begin{array}{rcl}
\neg\ A \wedge B \vee\ C & \equiv & ((\neg\ A) \wedge\ B) \vee\ C \\
s = t \wedge\ C & \equiv & (s = t) \wedge\ C \\
A \wedge\ B = B \wedge\ A & \equiv & A \wedge\ (B = B) \wedge\ A \\
\forall\, x.\ P\ x \wedge\ Q\ x & \equiv & \forall\, x.\ (P\ x \wedge\ Q\ x)
\end{array}
$$

Input syntax: $\longleftrightarrow$ (same priority as $\longrightarrow$)

Conventions:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \;\equiv\; A \wedge (B \wedge C)$

- $A \longrightarrow B \longrightarrow C \;\equiv\; A \longrightarrow (B \longrightarrow C)$
  $\not\equiv (A \longrightarrow B) \longrightarrow C$ **!**

- $\forall\, x\; y.\; P\; x\; y \;\equiv\; \forall\, x.\; \forall\, y.\; P\; x\; y \quad (\forall\,,\, \exists\,,\, \lambda,\, \ldots)$

Warning

Quantifiers have low priority
and need to be parenthesized (if in some context)

$$! \quad P \wedge \forall x.\ Q\ x \ \leadsto\ P \wedge (\forall x.\ Q\ x) \quad !$$

X-Symbols

... and their ascii representations:

<table>
<tr><td>$\forall$</td><td>\<forall></td><td>ALL</td></tr>
<tr><td>$\exists$</td><td>\<exists></td><td>EX</td></tr>
<tr><td>λ</td><td>\<lambda></td><td>%</td></tr>
<tr><td>$\longrightarrow$</td><td>--></td><td></td></tr>
<tr><td>$\longleftrightarrow$</td><td><--></td><td></td></tr>
<tr><td>$\wedge$</td><td>/\</td><td>&</td></tr>
<tr><td>$\vee$</td><td>\/</td><td>|</td></tr>
<tr><td>$\neg$</td><td>\<not></td><td>~</td></tr>
<tr><td>$\neq$</td><td>\<noteq></td><td>~=</td></tr>
</table>

Sets over type $'a$

$$'a\ set \quad = \quad 'a \Rightarrow bool$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-\ A$
- $\ldots$

$\in$	`\<in>`	`:`
$\subseteq$	`\<subseteq>`	`<=`
$\cup$	`\<union>`	`Un`
$\cap$	`\<inter>`	`Int`

6 Logic and Proof beyond "="

Logical Formulas

Proof Automation

Single Step Proofs

Inductive Definitions

simp and *auto*

simp: rewriting and a bit of arithmetic

auto: rewriting and a bit of arithmetic, logic and sets

- Show you where they got stuck
- highly incomplete
- Extensible with new *simp*-rules

Exception: *auto* acts on all subgoals

fastsimp

- rewriting, logic, sets, relations and a bit of arithmetic.
- incomplete but better than *auto*.
- Succeeds or fails
- Extensible with new *simp*-rules

blast

- A complete proof search procedure for FOL . . .
- . . . but (almost) without "$=$"
- Covers logic, sets and relations
- Succeeds or fails
- Extensible with new deduction rules

Automating arithmetic

arith:

- proves linear formulas (no "$*$")
- complete for quantifier-free *real* arithmetic
- complete for first-order theory of *nat* and *int* (Presburger arithmetic)

Sledgehammer

Architecture:

Isabelle

Formula & filtered library ↓ ↑ Proof = lemmas used

external **ATPs**[1]

Characteristics:

- Sometimes it works,
- sometimes it doesn't.

Do you feel lucky?

[1] Automatic Theorem Provers

by($\mathit{proof\text{-}method}$)

$$\approx$$

apply($\mathit{proof\text{-}method}$)
done

Auto_Proof_Demo.thy

6 Logic and Proof beyond "="

Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed.

Step-by-step proofs can occasionally be more readable than automagic proofs.

What are these *?-variables* ?

After you have finished a proof, Isabelle turns all free variables V in the theorem into $?V$.

Example: theorem `conjI`: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\leadsto$
  $[\![a = b; False]\!] \Longrightarrow a = b \wedge False$
- By unification:
  unifying $?P \wedge ?Q$ with $a{=}b \wedge False$
  sets $?P$ to $a{=}b$ and $?Q$ to $False$.

Rule application

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$
subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

Result: $1.\ \ldots \Longrightarrow A$
$2.\ \ldots \Longrightarrow B$

The general case: applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal $\ldots \Longrightarrow C$:

- Unify A and C
- Replace C with n new subgoals $A_1 \ldots A_n$

apply$(rule\ xyz)$

"Backchaining"

Typical backwards rules

$$\frac{?P \quad ?Q}{?P \wedge ?Q}\ \texttt{conjI}$$

$$\frac{?P \Longrightarrow ?Q}{?P \longrightarrow ?Q}\ \texttt{impI} \qquad \frac{\bigwedge x.\ ?P\ x}{\forall x.\ ?P\ x}\ \texttt{allI}$$

$$\frac{?P \Longrightarrow ?Q \quad ?Q \Longrightarrow ?P}{?P = ?Q}\ \texttt{iffI}$$

They are known as introduction rules
because they *introduce* a particular connective.

Teaching *blast* new intro rules

If r is a theorem $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ then

$$(\mathit{blast\ intro:\ r})$$

allows *blast* to backchain on r during proof search.

Example:

theorem trans: $[\![?x \leq ?y; ?y \leq ?z]\!] \Longrightarrow ?x \leq ?z$

goal $1.$ $[\![a \leq b; b \leq c; c \leq d]\!] \Longrightarrow a \leq d$

proof **apply**(*blast intro: trans*)

Can greatly increase the search space!

Forward proof: OF

If r is a theorem $\llbracket\ A_1; \ldots; A_n\ \rrbracket \Longrightarrow A$
and $r_1, \ldots, r_m$ $(m{\leq}n)$ are theorems then

$$r[OF\ r_1\ \ldots\ r_m]$$

is the theorem obtained
by proving $A_1\ \ldots\ A_m$ with $r_1\ \ldots\ r_m$.

Example: theorem `refl`: $?t = ?t$

```
conjI[OF refl[of "a"] refl[of "b"]]
```

$$\leadsto$$

$$a = a \wedge b = b$$

From now on: $?$ mostly suppressed on slides

Single_Step_Demo.thy

$\Longrightarrow$ versus $\longrightarrow$

$\Longrightarrow$ is part of the Isabelle framework. It structures theorems and proof states: $\llbracket\ A_1;\ \ldots;\ A_n\ \rrbracket \Longrightarrow A$

$\longrightarrow$ is part of HOL and can occur inside the logical formulas A_i and A.

Phrase theorems like this	$\llbracket\ A_1;\ \ldots;\ A_n\ \rrbracket \Longrightarrow A$
not like this	$A_1 \wedge \ldots \wedge A_n \longrightarrow A$

6 Logic and Proof beyond "="

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

inductive $Ev :: nat \Rightarrow bool$
where
  $Ev\ 0 \quad |$
  $Ev\ n \Longrightarrow Ev\ (n + 2)$

Easy proof: $Ev\ 4$

$$Ev\ 0 \Longrightarrow Ev\ 2 \Longrightarrow Ev\ 4$$

Trickier proof: $Ev\ m \Longrightarrow Ev\ (m+m)$

Idea: induction on the length of the proof of $Ev\ m$

Better: induction on the *structure* of the proof

Two cases: $Ev\ m$ is proved by

- rule $Ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow Ev\ (0+0)$
- rule $Ev\ n \Longrightarrow Ev\ (n+2)$
  $\Longrightarrow m = n+2$ and $Ev\ (n+n)$ (ind. hyp.!)
  $\Longrightarrow m+m = (n+2)+(n+2) = ((n+n)+2)+2$
  $\Longrightarrow Ev\ (m+m)$

Rule induction for Ev

To prove

$$Ev\ n \Longrightarrow P\ n$$

by *rule induction* on $Ev\ n$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule `Ev.induct`:

$$\frac{Ev\ n \quad P\ 0 \quad \bigwedge n.\ P\ n \Longrightarrow P(n+2)}{P\ n}$$

Format of inductive definitions

inductive $I :: \tau \Rightarrow bool$ **where**

$\llbracket\ I\ a_1;\ \ldots\ ;\ I\ a_n\ \rrbracket \Longrightarrow I\ a\ \ |$

$\vdots$

Note:

- I may have multiple arguments.
- Each rule may also contain *side conditions* not involving I.

Rule induction in general

To prove

$$I\ x \Longrightarrow P\ x$$

by *rule induction* on $I\ x$
we must prove for every rule

$$[\![\ I\ a_1; \ \dots\ ; I\ a_n\ ]\!] \Longrightarrow I\ a$$

that P is preserved:

$$[\![\ P\ a_1; \ \dots\ ; P\ a_n\ ]\!] \Longrightarrow P\ a$$

! Rule induction is absolutely central to (operational) semantics and the rest of this lecture course !

Inductive_Demo.thy

6 Logic and Proof beyond "="

7 Isar: A Language for Structured Proofs

Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!

Apply scripts versus Isar proofs

Apply script = assembly language program

Isar proof = structured program with comments

But: **apply** still useful for proof exploration

A typical Isar proof

proof
  assume $formula_0$
  have $formula_1$ **by** $simp$
  $\vdots$
  have $formula_n$ **by** $blast$
  show $formula_{n+1}$ **by** $\dots$
qed

proves $formula_0 \Longrightarrow formula_{n+1}$

Isar core syntax

proof = **proof** [method] step* **qed**
 | **by** method

method = $(simp \ldots) \mid (blast \ldots) \mid (induct \ldots) \mid \ldots$

step = **fix** variables ($\bigwedge$)
 | **assume** prop ($\Longrightarrow$)
 | [**from** fact$^+$] (**have** | **show**) prop proof

prop = [name:] "formula"

fact = name | ...

7 Isar: A Language for Structured Proofs

Example: Cantor's theorem

lemma $Cantor$: $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof default proof: assume $surj$, show $False$
  assume a: $surj\ f$
  from a **have** b: $\forall\ A.\ \exists\ a.\ A = f\ a$
    by$(simp\ add\text{: } surj\_def)$
  from b **have** c: $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$
    by $blast$
  from c **show** $False$
    by $blast$
qed

Isar_Demo.thy

Cantor and abbreviations

Abbreviations

$this$	$=$	the previous proposition proved or assumed
then	$=$	**from** $this$
thus	$=$	**then show**
hence	$=$	**then have**

using and **with**

(**have**|**show**) prop **using** facts

=

from facts (**have**|**show**) prop

with facts

=

from facts *this*

Structured lemma statement

lemma $Cantor'$:
  fixes $f :: 'a \Rightarrow 'a\ set$
  assumes s: $surj\ f$
  shows $False$
proof $-$ no automatic proof step
  have $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$ **using** s
    by$(auto\ simp{:}\ surj\_def)$
  thus $False$ **by** $blast$
qed

Proves $surj\ f \Longrightarrow False$
but $surj\ f$ *becomes local fact* s *in proof.*

The essence of structured proofs

Assumptions and intermediate facts
can be named and referred to explicitly and selectively

Structured lemma statements

fixes $x :: \tau_1$ **and** $y :: \tau_2$ $\ldots$
assumes *a:* P **and** *b:* Q $\ldots$
shows R

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**

7 Isar: A Language for Structured Proofs

$\longleftrightarrow$

show $P \longleftrightarrow Q$
proof
  assume P
  $\vdots$
  show Q $\ldots$
next
  assume Q
  $\vdots$
  show P $\ldots$
qed

Set equality and subset

show $A = B$
proof
  show $A \subseteq B$ $\ldots$
next
  show $B \subseteq A$ $\ldots$
qed

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\ldots$
qed

Case distinction

show R
proof $cases$
  assume P
  $\vdots$
  show R $\ldots$
next
  assume $\neg P$
  $\vdots$
  show R $\ldots$
qed

have $P \vee Q$ $\ldots$
then show R
proof
  assume P
  $\vdots$
  show R $\ldots$
next
  assume Q
  $\vdots$
  show R $\ldots$
qed

$\neg$

show $\neg\ P$
proof
  assume P
  $\vdots$
  show $False$ $\ldots$
qed

show P
proof $(rule\ ccontr)$
  assume $\neg P$
  $\vdots$
  show $False$ $\ldots$
qed

Contradiction

$\forall$ and $\exists$ introduction

show $\forall\, x.\ P(x)$
proof
  fix x local fixed variable
  show $P(x)$ $\ldots$
qed

show $\exists\, x.\ P(x)$
proof
  $\vdots$
  show $P(witness)$ $\ldots$
qed

$\exists$ elimination: **obtain**

have $\exists\, x.\; P(x)$
then obtain x **where** *p:* $P(x)$ **by** *blast*
$\vdots$ $\quad$ x *fixed local variable*

Works for one or more x

obtain example

lemma $Cantor''$: $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof
  assume $surj\ f$
  hence $\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**$(auto\ simp\!: surj\_def)$
  then obtain a **where** $\{x.\ x \notin f\ x\} = f\ a$ **by** $blast$
  hence $a \notin f\ a \longleftrightarrow a \in f\ a$ **by** $blast$
  thus $False$ **by** $blast$
qed

Isar_Demo.thy

Exercise

Example: pattern matching

show $formula_1 \longleftrightarrow formula_2$ (**is** $?L \longleftrightarrow ?R$)
proof
  assume $?L$
  $\vdots$
  show $?R$ ...
next
  assume $?R$
  $\vdots$
  show $?L$ ...
qed

$?thesis$

show $formula$ (is $?thesis$)
proof -
  ⋮
  show $?thesis$ ...
qed

Every show implicitly defines $?thesis$

let

Introducing local abbreviations in proofs:

let $?t$ = *"some-big-term"*
$\vdots$
have *"*$\ldots$ $?t$ $\ldots$*"*

Quoting facts by value

By name:

have *x0:* $''x > 0''$ ...
⋮
from *x0* ...

By value:

have $''x > 0''$ ...
⋮
from `x>0` ...

↑ ↑
back quotes

Isar_Demo.thy

Pattern matching and quotation

7 Isar: A Language for Structured Proofs

Example

lemma
assumes $xs = rev\ xs$
shows $(\exists\, ys.\ xs = ys\ @\ rev\ ys) \lor$
$(\exists\, ys\ a.\ xs = ys\ @\ a\ \#\ rev\ ys)$
proof ???

Isar_Demo.thy

Top down proof development

When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

have . . . **using** . . .
apply -   to make incoming facts part of proof state
apply *auto*   or whatever
apply . . .

At the end:

- **done**
- Better: convert to structured proof

7 Isar: A Language for Structured Proofs

moreover—ultimately

have P_1 ...		**have** lab_1: P_1 ...
moreover		**have** lab_2: P_2 ...
have P_2 ...		$\vdots$
moreover	$\approx$	**have** lab_n: P_n ...
$\vdots$		**from** lab_1 lab_2 ...
moreover		**have** P ...
have P_n ...		
ultimately		
have P ...		

With names

Raw proof blocks

$\{$ **fix** $x_1 \ \dots \ x_n$
assume $A_1 \ \dots \ A_m$
$\vdots$
have B
$\}$

proves $[\![\ A_1; \ \dots \ ; A_m \ ]\!] \Longrightarrow B$
where all x_i have been replaced by $?x_i$.

Isar_Demo.thy

moreover and { }

Proof state and Isar text

In general: **proof** *method*

Applies *method* and generates subgoal(s):

$$\bigwedge x_1 \ \dots \ x_n \ [\![\ A_1; \ \dots \ ; \ A_m \ ]\!] \Longrightarrow B$$

How to prove each subgoal:

fix $x_1 \ \dots \ x_n$
assume $A_1 \ \dots \ A_m$
$\vdots$
show B

Separated by **next**

7 Isar: A Language for Structured Proofs

Isar_Induct_Demo.thy

Case distinction

Datatype case distinction

datatype $t \; = \; C_1 \; \vec{\tau} \; \mid \; \ldots$

proof $(cases \; "term")$
  case $(C_1 \; \vec{x})$
  $\cdots \; \vec{x} \; \cdots$
next
$\vdots$
qed

where **case** $(C_i \; \vec{x}) \quad \equiv$

fix $\vec{x}$
assume $\underbrace{C_i\text{:}}_{\text{label}} \; \underbrace{term = (C_i \; \vec{x})}_{\text{formula}}$

Isar_Induct_Demo.thy

Structural induction for nat

Structural induction for nat

show $P(n)$
proof $(induct\ n)$
  case 0 $\equiv$ **let** $?case = P(0)$
  ⋮
  show $?case$
next
  case $(Suc\ n)$ $\equiv$ **fix** n **assume** Suc: $P(n)$
  ⋮ **let** $?case = P(Suc\ n)$
  ⋮
  show $?case$
qed

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$		
proof $(induct\ n)$		
case 0	$\equiv$	**fix** x **assume** 0: $A(0)$
$\vdots$		**let** $?case = P(0)$
show $?case$		
next		
case $(Suc\ n)$	$\equiv$	**fix** n
$\vdots$		**assume** Suc: $A(n) \Longrightarrow P(n)$ $A(Suc\ n)$
$\vdots$		**let** $?case = P(Suc\ n)$
show $?case$		
qed		

A remark on style

- **case** $(Suc\ n)$ … **show** $?case$
  is easy to write and maintain
- **fix** n **assume** $formula$ … **show** $formula'$
  is easier to read:
  - all information is shown locally
  - no contextual references (e.g. $?case$)

7 Isar: A Language for Structured Proofs

Isar_Induct_Demo.thy

Rule induction

Rule induction

inductive $I :: \tau \Rightarrow \sigma \Rightarrow bool$
where
$rule_1$: ...
⋮
$rule_n$: ...

show $I\ x\ y \Longrightarrow P\ x\ y$
proof (*induct rule: I.induct*)
  case $rule_1$
  ...
  show *?case*
next
⋮
next
  case $rule_n$
  ...
  show *?case*
qed

Fixing your own variable names

case $(rule_i\ x_1\ \ldots\ x_k)$

Renames the first k variables in $rule_i$ (from left to right) to $x_1\ \ldots\ x_k$.

The named assumptions

Given: an inductive definition of I.
In a proof of

$$I \ldots \Longrightarrow A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B,$$

in the context of

case R

we have

$R.hyps$ the assumptions of rule R, plus the induction hypothesis for each assumption $I \ldots$

$R.prems$ the premises A_i

$R = R.hyps \; @ \; R.prems$

Part IV

IMP: A Simple Imperative Language

Terminology

Statement: declaration of fact or claim

Semantics is easy.

Command: order to do something

Read the slides until you have understood them.

Expressions are evaluated, commands are executed

Commands

Concrete syntax:

$$
\begin{array}{rcl}
com & ::= & \texttt{SKIP} \\
& | & nat\ \texttt{::=}\ aexp \\
& | & com\ \texttt{;}\ com \\
& | & \texttt{IF}\ bexp\ \texttt{THEN}\ com\ \texttt{ELSE}\ com \\
& | & \texttt{WHILE}\ bexp\ \texttt{DO}\ com
\end{array}
$$

Commands

Abstract syntax:

$$
\begin{array}{lcl}
\textbf{datatype}\ com & = & SKIP \\
 & | & Assign\ nat\ aexp \\
 & | & Semi\ com\ com \\
 & | & If\ bexp\ com\ com \\
 & | & While\ bexp\ com
\end{array}
$$

Com.thy

Function update notation

If $f :: \tau_1 \Rightarrow \tau_2$ and $a :: \ \tau_1$ and $b :: \tau_2$ then

$$f(a := b)$$

is the function that behaves like f
except that it returns b for argument a.

$$f(a := b) = (\lambda x.\ \textit{if}\ x = a\ \textit{then}\ b\ \textit{else}\ f\ x)$$

8 IMP

Big Step Semantics

Small Step Semantics

Big step semantics

Concrete syntax:

$$(com,\ initial\text{-}state) \Rightarrow final\text{-}state$$

Intended meaning of $(c,\ s) \Rightarrow t$:

Command c started in state s terminates in state t

"$\Rightarrow$" here not type!

Big step rules

$$(SKIP,\ s) \Rightarrow s$$

$$(x ::= a,\ s) \Rightarrow s(x := aval\ a\ s)$$

$$\frac{(c_1,\ s_1) \Rightarrow s_2 \qquad (c_2,\ s_2) \Rightarrow s_3}{(c_1;\ c_2,\ s_1) \Rightarrow s_3}$$

Big step rules

$$\frac{bval\ b\ s \qquad (c_1,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

$$\frac{\neg\ bval\ b\ s \qquad (c_2,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

Big step rules

$$\frac{\neg\ bval\ b\ s}{(WHILE\ b\ DO\ c,\ s) \Rightarrow s}$$

$$\frac{(c,\ s_1) \Rightarrow s_2 \qquad \begin{array}{c} bval\ b\ s_1 \\ (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3 \end{array}}{(WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3}$$

Examples: derivation trees

$$\frac{\vdots}{(\mathit{0} ::= N\ \mathit{5};\ \mathit{1} ::= V\ \mathit{0},\ s) \Rightarrow\ \ ?} \qquad \frac{\vdots}{(w,\ s_i) \Rightarrow\ \ ?}$$

where
$$\begin{array}{lcl} w & = & \mathit{WHILE}\ b\ \mathit{DO}\ c \\ b & = & \mathit{NotEq}\ (V\ \mathit{0})\ (N\ \mathit{2}) \\ c & = & \mathit{0} ::= \mathit{Plus}\ (V\ \mathit{0})\ (N\ \mathit{1}) \end{array}$$

$\mathit{NotEq}\ a_1\ a_2 =$
$\mathit{Not}(\mathit{And}\ (\mathit{Not}(\mathit{Less}\ a_1\ a_2))\ (\mathit{Not}(\mathit{Less}\ a_2\ a_1)))$

and s_i is "$\{0 \mapsto i\}$" (formally: $s_i = \mathit{nth}\ [i]$).

Logically speaking

$$(c,\ s) \Rightarrow t$$

is just infix syntax for

$$big\_step\ (c,s)\ t$$

where

$$big\_step :: com \times state \Rightarrow state \Rightarrow bool$$

is an inductively defined predicate.

Big_Step.thy

Semantics

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$?
- $(x ::= a,\ s) \Rightarrow t$?
- $(c_1;\ c_2,\ s_1) \Rightarrow s_3$?

- $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$?

- $(w,\ s) \Rightarrow t$ where $w = WHILE\ b\ DO\ c$?

How are these inversions proved?
By case distinction:

> *Which rules could have derived* $(c, s) \Rightarrow t$*,*
> *and under which conditions?*

Automatic proof via **inductive_cases**
Produces an optimized format: *elimination rules*

We reformulate the inverted rules. Example:

$$\frac{(c_1;\ c_2,\ s_1) \Rightarrow s_3}{\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3}$$

is *logically equivalent* to the more convenient

$$\frac{(c_1;\ c_2,\ s_1) \Rightarrow s_3 \qquad \bigwedge s_2.\ [\![(c_1,\ s_1) \Rightarrow s_2;\ (c_2,\ s_2) \Rightarrow s_3]\!] \Longrightarrow P}{P}$$

Replaces assm $(c_1;\ c_2,\ s_1) \Rightarrow s_3$ by two assms $(c_1,\ s_1) \Rightarrow s_2$ and $(c_2,\ s_2) \Rightarrow s_3$ (with a new fixed s_2).
No $\exists$ and $\wedge$!

Similar for all other inverted rules

The general format: elimination rules

$$\frac{asm \quad asm_1 \Longrightarrow P \quad \dots \quad asm_n \Longrightarrow P}{P}$$

(possibly with $\bigwedge \overline{x}$ in front of the $asm_i \Longlongrightarrow P$)

Reading:

To prove a goal P with assumption asm,
prove all $asm_i \Longrightarrow P$

Example:

$$\frac{F \ \vee \ G \quad F \ \Longrightarrow \ P \quad G \ \Longrightarrow \ P}{P}$$

elim attribute

- Theorems with *elim* attribute are used automatically by *blast*, *fastsimp* and *auto*
- Can also be added locally, eg *(blast elim: . . .)*
- Variant: *elim!* applies elim-rules eagerly.

Big_Step.thy

Rule inversion

Command equivalence

Two commands have the same input/output behaviour:

$$c \sim c' \;\equiv\; (\forall\, s\; t.\; (c,s) \Rightarrow t \longleftrightarrow (c',s) \Rightarrow t)$$

Example

$$w \sim iw$$

where $w = \textit{WHILE } b \textit{ DO } c$
$iw = \textit{IF } b \textit{ THEN } c;\ w \textit{ ELSE SKIP}$

A derivation-based proof:
transform any derivation of $(w,\ s) \Rightarrow t$
into a derivation of $(iw,\ s) \Rightarrow t$,
and vice versa.

A formula-based proof

$$(w,\ s) \Rightarrow\ t$$

$$\longleftrightarrow$$

$$bval\ b\ s \wedge (\exists\, s'.\ (c,\ s) \Rightarrow\ s' \wedge (w,\ s') \Rightarrow\ t)$$
$$\vee$$
$$\neg\ bval\ b\ s \wedge t =\ s$$

$$\longleftrightarrow$$

$$(iw,\ s) \Rightarrow\ t$$

Using the rules and rule inversions for $\Rightarrow$.

Big_Step.thy

Command equivalence

Execution is deterministic

Any two executions of the same command in the same start state lead to the same final state:

$$(c,\ s) \Rightarrow t \implies (c,\ s) \Rightarrow t' \implies t = t'$$

Proof by rule induction, for arbitrary t'.

Big_Step.thy

Execution is deterministic

The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

$$(c,s) \text{ does not terminate iff } \nexists\, t.\ (c,\ s) \Rightarrow\ t\ ?$$

Needs a formal notion of nontermination to prove it.
Could be wrong if we have forgotten a $\Rightarrow$ rule.

Big step semantics cannot directly describe

- nonterminating computations,
- parallel computations.

We need a finer grained semantics!

8 IMP

Big Step Semantics

Small Step Semantics

Small step semantics

Concrete syntax:

$$(com, state) \to (com, state)$$

Intended meaning of $(c,\ s) \to (c',\ s')$:

> *The first step in the execution of c in state s*
> *leaves a "remainder" command c'*
> *to be executed in state s'.*

Execution as finite or infinite reduction:

$$(c_1, s_1) \to (c_2, s_2) \to (c_3, s_3) \to \ldots$$

Terminology

- A pair (c,s) is called a configuration.
- If $cs \to cs'$ we say that cs reduces to cs'.
- A configuration cs is final iff $\nexists\, cs'.\ cs \to cs'$

The intention:

$$(SKIP,\ s)\ \text{ is final}$$

Why?

$SKIP$ is the empty program. Nothing more to be done.

Small step rules

$$(x ::= a,\ s) \ \rightarrow\ (SKIP,\ s(x := \ aval\ a\ s))$$

$$(SKIP;\ c,\ s) \ \rightarrow\ (c,\ s)$$

$$\frac{(c_1, s) \ \rightarrow\ (c_1', s')}{(c_1; c_2, s) \ \rightarrow\ (c_1'; c_2, s')}$$

Small step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_1, s)}$$

$$\frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_2, s)}$$

$$(WHILE\ b\ DO\ c,\ s) \ \rightarrow$$
$$(IF\ b\ THEN\ c;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s)$$

Fact $(SKIP, s)$ is a final configuration.

Small step examples

$$(2 ::= V\ 0;\ 0 ::= V\ 1;\ 1 ::= V\ 2,\ s) \to \ldots$$

where $s = nth\ [11,\ 13,\ 17]$.

$$(w,\ s_0) \to \ldots$$

where
$$\begin{aligned} w &= WHILE\ b\ DO\ c \\ b &= Less\ (V\ 0)\ (N\ 1) \\ c &= 0 ::= Plus\ (V\ 0)\ (N\ 1) \\ s_n &= nth\ [n] \end{aligned}$$

`Small_Step.thy`

Semantics

Are big and small step semantics equivalent?

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP, t)$

Proof by rule induction (of course on $cs \Rightarrow t$)

From $\rightarrow*$ to $\Rightarrow$

Theorem $cs \rightarrow* (SKIP, t) \implies cs \Rightarrow t$

Needs to be generalized:

Lemma 1 $cs \rightarrow* cs' \implies cs' \Rightarrow t \implies cs \Rightarrow t$

Now Theorem follows from Lemma 1 by $(SKIP, t) \Rightarrow t$.

Lemma 1 is proved by rule induction on $cs \rightarrow* cs'$.
Needs

Lemma 2 $cs \rightarrow cs' \implies cs' \Rightarrow t \implies cs \Rightarrow t$

Lemma 2 is proved by rule induction on $cs \rightarrow cs'$.

Equivalence

Corollary $cs \Rightarrow t \longleftrightarrow cs \rightarrow* (SKIP, t)$

Small_Step.thy

Equivalence of big and small

Can execution stop prematurely?

That is, are there any final configs except $(SKIP,s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive ($c \neq SKIP \Longrightarrow \neg$ $\mathit{final}(c,s)$)
by induction on c.

- Case $c_1;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;\ c_2,\ s)$
  - $c_1 \neq SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1,\ s)$ (by IH) $\Longrightarrow \neg\ \mathit{final}\ (c_1;\ c_2,\ s)$
- Remaining cases: trivial or easy

By rule inversion: $(SKIP, s) \to ct \Longrightarrow False$

Together:

Corollary $\mathit{final}\ (c, s) = (c = SKIP)$

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow t) = (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow t)$

$= \ (\exists\, t.\ cs \rightarrow\!* \ (\mathit{SKIP},t))$

(by big = small)

$= \ (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs')$

(by final = SKIP)

Equivalent:

$\Rightarrow$ does not yield final state iff $\rightarrow$ does not terminate

May versus Must

$\to$ is deterministic:

Lemma $cs \to cs' \implies cs \to cs'' \implies cs'' = cs'$
(Proof by rule induction)

Therefore: no difference between

may terminate (there is a terminating $\to$ path)

must terminate (all $\to$ paths terminate)

Therefore: $\Rightarrow$ correctly reflects termination behaviour.

With nondeterminism: may have both $cs \Rightarrow t$ and a nonterminating reduction $cs \to cs' \to \ldots$

9 Compiler

Stack Machine

Compiler

Stack Machine

Instructions:

datatype $instr =$

$PUSH\_N\ nat$	
$PUSH\_V\ nat$	
$ADD \mid$	
$STORE\ nat \mid$	store
$JMPF\ nat \mid$	jump fwd
$JMPB\ nat \mid$	jump bwd
$JMPFLESS\ nat \mid$	jump fwd if $<$
$JMPFGE\ nat$	jump fwd if $\geq$

Type abbreviations:

$$\begin{array}{rcl} stack & = & nat\ list \\ config & = & nat \times state \times stack \end{array}$$

Execution of 1 instruction:

$$P \vdash (pc,\ s,\ stk) \rightarrow (pc',\ s',\ stk')$$

$$instr\ list \vdash config \rightarrow config$$

$$\frac{i < |P| \qquad P \,!\, i = PUSH\_N\ n}{P \vdash (i,\ s,\ stk) \rightarrow (i + 1,\ s,\ n \# stk)}$$

$$\frac{i < |P| \qquad P \,!\, i = PUSH\_V\ x}{P \vdash (i,\ s,\ stk) \rightarrow (i + 1,\ s,\ s\ x \# stk)}$$

$$\frac{i < |P| \qquad P \,!\, i = ADD}{P \vdash (i,\ s,\ stk) \rightarrow (i + 1,\ s,\ (hd2\ stk + hd\ stk) \# tl2\ stk)}$$

$$\frac{i < |P| \qquad P \,!\, i = STORE\ n}{P \vdash (i,\ s,\ stk) \rightarrow (i + 1,\ s(n := hd\ stk),\ tl\ stk)}$$

$$\frac{i < |P| \qquad P \; ! \; i = \mathit{JMPF} \; n}{P \vdash (i,\ s,\ stk) \rightarrow (i + 1 + n,\ s,\ stk)}$$

$$\frac{i < |P| \qquad P \; ! \; i = \mathit{JMPB} \; n \qquad n \leq i + 1}{P \vdash (i,\ s,\ stk) \rightarrow (i + 1 - n,\ s,\ stk)}$$

$$\frac{i < |P| \qquad P \; ! \; i = \mathit{JMPFLESS} \; n}{P \vdash (i,\ s,\ stk) \rightarrow (i',\ s,\ tl2 \; stk)}$$

where

$i' = (\textsf{\textit{if}}\ hd2\ stk < hd\ stk\ \textsf{\textit{then}}\ i + 1 + n\ \textsf{\textit{else}}\ i + 1)$

JMPFGE: analogous

Defined in the usual manner:

$$P \vdash (pc,\ s,\ stk) \rightarrow\!* \ (pc',\ s',\ stk')$$

Compiler.thy

Stack Machine

9 Compiler

Stack Machine

Compiler

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [PUSH\_N\ n]$$
$$acomp\ (V\ n) = [PUSH\_V\ n]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

$$acomp\ a \vdash (0,\ s,\ stk) \rightarrow* (|acomp\ a|,\ s,\ aval\ a\ s\ \#\ stk)$$

Proof by induction on a (with arbitrary stk).

Needs lemmas!

$$P \vdash c \rightarrow* c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow* c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow* (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (|P'| + i,\ s,\ stk) \rightarrow* (|P'| + i',\ s',\ stk')$$

Proofs by rule induction on $\rightarrow*$,
using the corresponding single step lemmas:

$$P \vdash c \rightarrow c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (|P'| + i,\ s,\ stk) \rightarrow (|P'| + i',\ s',\ stk')$$

Proofs by cases/induction.

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

- Either execution leads to the end of ins
- or it jumps to offset $+n$ beyond ins.

Parameters: when to jump (if b is $True$ or $False$)
where to jump to (n)

$$bcomp :: bexp \Rightarrow bool \Rightarrow nat \Rightarrow instr\ list$$

Example

Let $b =$
$And\ (Less\ (V\ 0)\ (V\ 1))\ (Not\ (Less\ (V\ 2)\ (V\ 3)))$.

$bcomp\ b\ False\ 3 =$

$[PUSH\_V\ 0$,
$PUSH\_V\ 1$,

,
$PUSH\_V\ 2$,
$PUSH\_V\ 3$,

$]$

$$bcomp :: bexp \Rightarrow bool \Rightarrow nat \Rightarrow instr\ list$$

$bcomp\ (B\ v)\ c\ n = (\textsf{if}\ v = c\ \textsf{then}\ [JMPF\ n]\ \textsf{else}\ [])$

$bcomp\ (Not\ b)\ c\ n = bcomp\ b\ (\neg c)\ n$

$bcomp\ (Less\ a_1\ a_2)\ c\ n =$

$acomp\ a_1\ @$
$acomp\ a_2\ @$
$(\textsf{if}\ c\ \textsf{then}\ [JMPFLESS\ n]\ \textsf{else}\ [JMPFGE\ n])$

$bcomp\ (And\ b_1\ b_2)\ c\ n =$

$\textsf{let}\ cb_2 = bcomp\ b_2\ c\ n;$
$\quad m = \textsf{if}\ c\ \textsf{then}\ |cb_2|\ \textsf{else}\ |cb_2| + n;$
$\quad cb_1 = bcomp\ b_1\ False\ m$
$\textsf{in}\ cb_1\ @\ cb_2$

Correctness of $bcomp$

$$
\begin{array}{l}
bcomp\ b\ c\ n \\
\vdash (0,\ s,\ stk) \rightarrow * \\
\quad (|bcomp\ b\ c\ n| + (\textit{if}\ c = bval\ b\ s\ \textit{then}\ n\ \textit{else}\ 0),\ s, \\
\quad\ stk)
\end{array}
$$

Compiling *com*

$ccomp :: com \Rightarrow instr\ list$

$ccomp\ SKIP = []$

$ccomp\ (x ::= a) = acomp\ a\ @\ [STORE\ x]$

$ccomp\ (c_1;\ c_2) = ccomp\ c_1\ @\ ccomp\ c_2$

$$ccomp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$$

$$\textsf{let}\ cc_1 = ccomp\ c_1;\ cc_2 = ccomp\ c_2;$$
$$\quad cb = bcomp\ b\ False\ (|cc_1| + 1)$$
$$\textsf{in}\ cb\ @\ cc_1\ @\ JMPF\ |cc_2|\ \#\ cc_2$$

$$ccomp\ (WHILE\ b\ DO\ c) =$$

$$\textsf{let}\ cc = ccomp\ c;\ cb = bcomp\ b\ False\ (|cc| + 1)$$
$$\textsf{in}\ cb\ @\ cc\ @\ [JMPB\ (|cb| + |cc| + 1)]$$

Correctness of $ccomp$

If the source code produces a certain result,
so should the compiled code:

$$(c,\ s) \Rightarrow t \Longrightarrow$$
$$ccomp\ c \vdash (\mathit{0},\ s,\ stk) \rightarrow* (|ccomp\ c|,\ t,\ stk)$$

Proof by rule induction.

The other direction

We have only shown

compiled code simulates source code.

How about $\Longleftarrow$:

source code simulates compiled code?

If $ccomp\ c$ produces result t, and if $(c,\ s) \Rightarrow t'$, then $\Longrightarrow$ implies that $ccomp\ c$ must also produce t' and thus $t' = t$ (why?).

But we have *not* ruled out this potential error:

c does not terminate but $ccomp\ c$ does.

We stop here.

10 A Typed Version of IMP

Why Types?

To prevent mistakes, dummy!

There are 3 kinds of types

The Good
: Static types that *guarantee* absence of certain runtime faults.
Example: no memory access errors in Java.

The Bad
: Static types that have mostly decorative value but do not guarantee anything at runtime.
Example: C, C++

The Ugly
: Dynamic types that detect errors when it can be too late.
Example: "Message not understood" in Smalltalk.

The ideal

Well-typed programs cannot go wrong.

Robin Milner, *A Theory of Type Polymorphism in Programming*, 1978.

The most influential slogan and one of the most influential papers in programming language theory.

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination
4. Run out of memory
5. Secret leaked
6. and many more ...

There are type systems for *everything* (and more) but in practice (Java, C#) only 1 is covered.

Type safety

A programming language is type safe if the execution of a well-typed program cannot lead to certain errors.

Java and the JVM have been *proved* to be type safe. (Note: Java exceptions are not errors!)

Correctness and completeness

Type soundness means that the type system is sound/correct w.r.t. the semantics:

> *If the type system says yes,*
> *the semantics does not lead to an error.*

The semantics is the primary definition,
the type system must be justified w.r.t. it.

How about completeness? Remember Rice:

> *Nontrivial semantic properties of programs*
> *(e.g. termination) are undecidable.*

Hence there is no (decidable) type system that accepts *all* programs that have a certain semantic property.

Automatic analysis of semantic program properties
is necessarily incomplete.

10 A Typed Version of IMP

Arithmetic

Values:

datatype $val = Iv\ int \mid Rv\ real$

The state:

$state = name \Rightarrow val$

Arithmetic expresssions:

datatype $aexp =$
$\quad Ic\ int \mid Rc\ real \mid V\ name \mid Plus\ aexp\ aexp$

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!
No automatic coercions.

Does this mean any implementation of IMP also needs to tag values?

No! Compilers compile only well-typed programs, and well-typed programs do not need tags.

Tags are only used to detect certain errors
and to prove that the type system avoids those errors.

Evaluation of $aexp$

Not recursively function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

$$taval\ (Ic\ i)\ s\ (Iv\ i)$$

$$taval\ (Rc\ r)\ s\ (Rv\ r)$$

$$taval\ (V\ x)\ s\ (s\ x)$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Iv\ (i_1 + i_2))}$$

$$\frac{taval\ a_1\ s\ (Rv\ r_1) \qquad taval\ a_2\ s\ (Rv\ r_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Rv\ (r_1 + r_2))}$$

Example: evaluation of $Plus\ (V\ 0)\ (Ic\ 1)$

If $s\ 0 = Iv\ i$:

$$\frac{taval\ (V\ 0)\ s\ (Iv\ i) \qquad taval\ (Ic\ 1)\ s\ (Iv\ 1)}{taval\ (Plus\ (V\ 0)\ (Ic\ 1))\ s\ (Iv(i + 1))}$$

If $s\ 0 = Rv\ r$: then there is *no* value v such that $taval\ (Plus\ (V\ 0)\ (Ic\ 1))\ s\ v$.

The functional alternative

An extremely useful datatype:

datatype $'a\ option = None \mid Some\ 'a$

A "partial" function:

$$taval :: aexp \Rightarrow state \Rightarrow val\ option$$

Exercise!

Boolean expressions

Defined as usual.

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

$$tbval\ (B\ bv)\ s\ bv \qquad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)}$$

$$\frac{tbval\ b_1\ s\ bv_1 \qquad tbval\ b_2\ s\ bv_2}{tbval\ (And\ b_1\ b_2)\ s\ (bv_1 \wedge bv_2)}$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (i_1 < i_2)}$$

$$\frac{taval\ a_1\ s\ (Rv\ r_1) \qquad taval\ a_2\ s\ (Rv\ r_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (r_1 < r_2)}$$

com: big or small steps?

We need to detect if things “go wrong”.

- Big step semantics:
  Cannot model error by absence of final state.
  Would confuse error and nontermination.
  Could introduce an extra error-element, e.g.
  $big\_step :: com \times state \Rightarrow state\ option \Rightarrow bool$
  Complicates formalization.
- Small step semantics:
  error = semantics gets stuck

Small step semantics

$$\frac{taval\ a\ s\ v}{(x ::= a,\ s) \rightarrow (SKIP,\ s(x := v))}$$

$$\frac{tbval\ b\ s\ True}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \rightarrow (c_1,\ s)}$$

$$\frac{tbval\ b\ s\ False}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \rightarrow (c_2,\ s)}$$

The other rules remain unchanged.

Example

Let $c \;=\; (x ::= \mathit{Plus}\ (V\ \mathit{0})\ (\mathit{Ic}\ \mathit{1}))$.

- If $s\ \mathit{0} = \mathit{Iv}\ i$: $(c,\ s) \rightarrow (\mathit{SKIP},\ s(x := \mathit{Iv}\ (i + \mathit{1})))$
- If $s\ \mathit{0} = \mathit{Rv}\ r$: $(c,\ s) \not\rightarrow$

10 A Typed Version of IMP

Type system

There are two types:

datatype $ty = Ity \mid Rty$

What is the type of $Plus\ (V\ 0)\ (V\ 1)$?

Depends on the type of $V\ 0$ and $V\ 1$!

A type environment maps variable names to their types:
$tyenv = name \Rightarrow ty$

The type of an expression is always *relative to / in the context of* a type enviroment Γ. Standard notation:

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

The type of an $aexp$

$$\Gamma \vdash a : \tau$$
$$tyenv \vdash aexp : ty$$

The rules:

$$\Gamma \vdash Ic\ i : Ity$$

$$\Gamma \vdash Rc\ r : Rty$$

$$\Gamma \vdash V\ x : \Gamma\ x$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Plus\ a_1\ a_2 : \tau}$$

Example

$$\frac{\vdots}{\Gamma \vdash \mathit{Plus}\ (V\ \mathit{0})\ (\mathit{Plus}\ (V\ \mathit{0})\ (\mathit{Ic}\ \mathit{0}))\ :\ \mathit{?}}$$

where $\Gamma\ \mathit{0} = \mathit{Ity}$.

Well-typed $bexp$

Notation:

$$\Gamma \vdash b$$
$$tyenv \vdash bexp$$

Read: In context Γ, b is well-typed.

The rules:

$$\Gamma \vdash B\ bv$$

$$\frac{\Gamma \vdash b}{\Gamma \vdash Not\ b}$$

$$\frac{\Gamma \vdash b_1 \qquad \Gamma \vdash b_2}{\Gamma \vdash And\ b_1\ b_2}$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Less\ a_1\ a_2}$$

Example: $\Gamma \vdash Less\ (Ic\ i)\ (Rc\ r)$

Well-typed commands

Notation:

$$\Gamma \vdash c$$
$$tyenv \vdash com$$

Read: In context Γ, c is well-typed.

The rules:

$$\Gamma \vdash \mathit{SKIP}$$

$$\frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

$$\frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1;\ c_2}$$

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2}$$

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash c}{\Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

Interlude: Rule formats

Let $P(t)$ be an inductively defined predicate (e.g. well-typedness) such that

- t is of some syntactic type (e.g. $aexp$), i.e. some datatype, and
- the definition is executable, i.e. the output (e.g. the type) is computable from the input t by backchaining.

All our semantics and type systems have this property.

The definition of P is

- syntax directed if there is exactly one rule for each syntactic construct.

  $\Longrightarrow$ no backtracking needed during execution

- compositional if $P(c\ t_1 \ldots t_n)$ depends only on $P(t_1)$, ..., $P(t_n)$.

  $\Longrightarrow$ execution always terminates (if the rules do not use other nonterminating predicates)

$\Longrightarrow$ A syntax directed, compositional definition of $P(t)$ allows execution in $|t|$ many backchaining steps.

$$\frac{A_1 \qquad \dots \qquad A_n}{B}$$

is invertible if

$$\frac{B}{A_1 \wedge \ \dots \ \wedge A_n}$$

also holds.

Which of our type systems consist only of invertible rules?

A syntax directed, compositional definition which consists only of invertible rules can be defined as a recursive function by considering each rule as an equation.

A recursive definition of $\Gamma \vdash c$

$$
\begin{array}{lcl}
\Gamma \vdash \mathit{SKIP} & \longleftrightarrow & \mathit{True} \\
\Gamma \vdash x ::= a & \longleftrightarrow & \Gamma \vdash a : \Gamma\ x \\
\Gamma \vdash c_1;\ c_2 & \longleftrightarrow & \Gamma \vdash c_1 \land \Gamma \vdash c_2 \\
\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2 & \longleftrightarrow & \\
\quad \Gamma \vdash b \land \Gamma \vdash c_1 \land \Gamma \vdash c_2 & & \\
\Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c & \longleftrightarrow & \Gamma \vdash b \land \Gamma \vdash c
\end{array}
$$

Is easier to use than traditional inductive one.

10 A Typed Version of IMP

Well-typed states

Even well-typed programs can get stuck ...
... if they start in a bad state.

Remember:
If $s\ 0 = Rv\ r$ then $(x ::= Plus\ (V\ 0)\ (Ic\ 1),\ s) \not\rightarrow$

The state must be well-typed w.r.t. Γ.

Frequent alternative terminology:
The state must conform to Γ.

The type of a value:

$$type\ (Iv\ i) = Ity$$
$$type\ (Rv\ r) = Rty$$

Well-typed state:

$$(\Gamma \vdash s) = (\forall x.\ type\ (s\ x) = \Gamma\ x)$$

Type soundness

Reduction cannot get stuck:

> *If everything is ok (* $\Gamma \vdash s$, $\Gamma \vdash c$ *),*
> *and you take a finite number of steps,*
> *and you have not reached SKIP,*
> *then you can take one more step.*

Follows from progress:

> *If everything is ok and you have not reached SKIP,*
> *then you can take one more step.*

and preservation:

> *If everything is ok and you take a step,*
> *then everything is ok again.*

The slogan

$$\text{Progress} \wedge \text{Preservation} \implies \text{Type safety}$$

Progress	Well-typed programs do not get stuck.
Preservation	Well-typedness is preserved by reduction.

Preservation: Well-typedness is an *invariant*.

com

Progress:

$$[\![\Gamma \vdash c;\ \Gamma \vdash s;\ c \neq SKIP]\!] \Longrightarrow \exists\, cs'.\ (c,\ s) \to cs'$$

Preservation:

$$[\![(c,\ s) \to (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s]\!] \Longrightarrow \Gamma \vdash s'$$

$$[\![(c,\ s) \to (c',\ s');\ \Gamma \vdash c]\!] \Longrightarrow \Gamma \vdash c'$$

Type soundness:

$$[\![(c,\ s) \to\!* \ (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s;\ c' \neq SKIP]\!]$$
$$\Longrightarrow \exists\, cs''.\ (c',\ s') \to cs''$$

bexp

Progress:

$$[\![\Gamma \vdash b;\ \Gamma \vdash s]\!] \Longrightarrow \exists\, v.\ tbval\ b\ s\ v$$

$aexp$

Progress:

$$[\![\Gamma \vdash a : \tau;\ \Gamma \vdash s]\!] \Longrightarrow \exists\, v.\ taval\ a\ s\ v$$

Preservation:

$$[\![\Gamma \vdash a : \tau;\ taval\ a\ s\ v;\ \Gamma \vdash s]\!] \Longrightarrow type\ v = \tau$$

All proofs by rule induction.

Types.thy

The mantra

Type systems have a purpose:

The static analysis of programs
in order to predict their runtime behaviour.

The correctness of the prediction must be provable.

Part V

Data-Flow Analyses and Optimization

Each local variable must have a definitely assigned value when any access of its value occurs. A compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable x*,* x *is definitely assigned before the access; otherwise a compile-time error must occur.*

Java Language Specification

Java was the first language to force programmers to initialize their variables.

Java versus IMP:

- Java has local variables and parameters;
  parameters are always initialized.
- IMP: we assume that certain variables are initialized before the program starts.

Examples: ok or not?

Assume x is initialized and $x \neq y$.

$$\mathit{IF}\ \mathit{Less}\ (V\ x)\ (N\ \mathit{1})\ \mathit{THEN}\ y ::= V\ x$$
$$\mathit{ELSE}\ y ::= \mathit{Plus}\ (V\ x)\ (N\ \mathit{1});$$
$$y ::= \mathit{Plus}\ (V\ y)\ (N\ \mathit{1})$$

$$\mathit{IF}\ \mathit{Less}\ (V\ x)\ (V\ x)\ \mathit{THEN}\ y ::= \mathit{Plus}\ (V\ y)\ (N\ \mathit{1})$$
$$\mathit{ELSE}\ y ::= V\ x$$

Assume x and y are initialized and $\mathit{distinct}\ [x,\ y,\ z]$:

$$\mathit{WHILE}\ \mathit{Less}\ (V\ x)\ (V\ y)\ \mathit{DO}\ z ::= V\ x;$$
$$z ::= \mathit{Plus}\ (V\ z)\ (N\ \mathit{1})$$

Simplifying principle

We do not analyze boolean expressions to determine program execution.

11 Definite Assignment Analysis

Prelude: Variables in Expressions

Definite Assignment Analysis

Initialization Sensitive Semantics

Theory $Vars$ provides an overloaded function $vars$:

$vars :: aexp \Rightarrow name\ set$
$vars\ (N\ n) = \emptyset$
$vars\ (V\ x) = \{x\}$
$vars\ (Plus\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

$vars :: bexp \Rightarrow name\ set$
$vars\ (B\ bv) = \emptyset$
$vars\ (Not\ b) = vars\ b$
$vars\ (And\ b_1\ b_2) = vars\ b_1 \cup vars\ b_2$
$vars\ (Less\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

Vars.thy

Modified example from the JLS:

> *Variable x is definitely assigned after $SKIP$*
> *iff x is definitely assigned before $SKIP$.*

Similar statements for each each language construct.

$D :: name\ set \Rightarrow com \Rightarrow name\ set \Rightarrow bool$

$D\ A\ c\ A'$ should imply:

> *If all variables in A are initialized before c is executed,*
> *then no uninitialized variable is accessed during execution,*
> *and all variables in A' are initialized afterwards.*

$$D\ A\ SKIP\ A$$

$$\frac{vars\ a \subseteq A}{D\ A\ (x ::= a)\ (\{x\} \cup A)}$$

$$\frac{D\ A_1\ c_1\ A_2 \qquad D\ A_2\ c_2\ A_3}{D\ A_1\ (c_1;\ c_2)\ A_3}$$

$$\frac{vars\ b \subseteq A \qquad D\ A\ c_1\ A_1 \qquad D\ A\ c_2\ A_2}{D\ A\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ (A_1 \cap A_2)}$$

$$\frac{vars\ b \subseteq A \qquad D\ A\ c\ A'}{D\ A\ (WHILE\ b\ DO\ c)\ A}$$

Correctness of D

- Things can go wrong:
  execution may access uninitialized variable.

  $\Longrightarrow$ We need a new, finger grained semantics.

- Big step semantics:
  semantics longer, correctness proof shorter

- Small step semantics:
  semantics shorter, correctness proof longer

For variety's sake, we choose a big step semantics.

11 Definite Assignment Analysis

Prelude: Variables in Expressions

Definite Assignment Analysis

Initialization Sensitive Semantics

$$state = name \Rightarrow nat\ option$$

where

datatype $'a\ option = None \mid Some\ 'a$

Notation: $s(x \mapsto y)$ means $s(x := Some\ y)$

Definition: $dom\ s = \{a \mid s\ a \neq None\}$

Expression evaluation

$aval :: aexp \Rightarrow state \Rightarrow val\ option$

$aval\ (N\ i)\ s = Some\ i$

$aval\ (V\ x)\ s = s\ x$

$$
\begin{array}{l}
aval\ (Plus\ a_1\ a_2)\ s = \\
(\textsf{case}\ (aval\ a_1\ s,\ aval\ a_2\ s)\ \textsf{of} \\
\quad (Some\ i_1,\ Some\ i_2) \Rightarrow Some(i_1{+}i_2) \\
\ |\ \_ \Rightarrow None)
\end{array}
$$

$bval :: bexp \Rightarrow state \Rightarrow bool\ option$

$bval\ (B\ bv)\ s = Some\ bv$

$bval\ (Not\ b)\ s =$
$(\textsf{case}\ bval\ b\ s\ \textsf{of}\ None \Rightarrow None$
$\mid Some\ bv \Rightarrow Some\ (\neg\ bv))$

$bval\ (And\ b_1\ b_2)\ s =$
$(\textsf{case}\ (bval\ b_1\ s,\ bval\ b_2\ s)\ \textsf{of}$
$(Some\ bv_1,\ Some\ bv_2) \Rightarrow Some(bv_1 \wedge bv_2)$
$\mid \_ \Rightarrow None)$
$bval\ (Less\ a_1\ a_2)\ s =$
$(\textsf{case}\ (aval\ a_1\ s,\ aval\ a_2\ s)\ \textsf{of}$
$(Some\ i_1,\ Some\ i_2) \Rightarrow Some(i_1 < i_2)$
$\mid \_ \Rightarrow None)$

Big step semantics

$$(com,\ state) \Rightarrow state\ option$$

A small complication:

$$\frac{(c_1,\ s_1) \Rightarrow Some\ s_2 \qquad (c_2,\ s_2) \Rightarrow s}{(c_1;\ c_2,\ s_1) \Rightarrow s}$$

$$\frac{(c_1,\ s_1) \Rightarrow None}{(c_1;\ c_2,\ s_1) \Rightarrow None}$$

More convenient, because compositional:

$$(com,\ state\ option) \Rightarrow state\ option$$

Error ($None$) propagates:

$$(c,\ None) \Rightarrow None$$

Execution starting in (mostly) normal states ($Some\ s$):

$$(SKIP,\ s) \Rightarrow s$$

$$\frac{aval\ a\ s = Some\ i}{(x ::= a,\ Some\ s) \Rightarrow Some\ (s(x \mapsto i))}$$

$$\frac{aval\ a\ s = None}{(x ::= a,\ Some\ s) \Rightarrow None}$$

$$\frac{(c_1,\ s_1) \Rightarrow s_2 \qquad (c_2,\ s_2) \Rightarrow s_3}{(c_1;\ c_2,\ s_1) \Rightarrow s_3}$$

$$\frac{bval\ b\ s = Some\ True \qquad (c_1,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ False \qquad (c_2,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = None}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow None}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{bval\ b\ s = Some\ True \qquad (c,\ Some\ s) \Rightarrow s' \qquad (WHILE\ b\ DO\ c,\ s') \Rightarrow s''}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

$$\frac{bval\ b\ s = None}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow None}$$

Correctness of D w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

$\textit{If } D\ (\textit{dom } s)\ c\ A' \textit{ and } (c,\ \textit{Some } s) \Rightarrow s'$
$\textit{then } s' \neq \textit{None}.$

We need to prove a generalized statement:

$\textit{If } (c,\ \textit{Some } s) \Rightarrow s' \textit{ and } D\ A\ c\ A' \textit{ and } A \subseteq \textit{dom } s$
$\textit{then } \exists\, t.\ s' = \textit{Some } t \wedge A' \subseteq \textit{dom } t.$

By rule induction on $(c,\ \textit{Some } s) \Rightarrow s'$.

Proof needs some easy lemmas:

$$vars\ a \subseteq\ dom\ s \Longrightarrow \exists\, i.\ aval\ a\ s = Some\ i$$

$$vars\ b \subseteq\ dom\ s \Longrightarrow \exists\, bv.\ bval\ b\ s = Some\ bv$$

$$D\ A\ c\ A' \Longrightarrow A \subseteq A'$$

Motivation

Consider the following program (where $x \neq y$):

$$x ::= Plus\ (V\ y)\ (N\ 1);$$
$$y ::= N\ 5;$$
$$x ::= Plus\ (V\ y)\ (N\ 3)$$

The first assignment is redundant and can be removed because x is dead at that point.

Semantically, a variable x is live before command c
if the initial value of x can influence the final state.

As a sufficient condition, we call x live before c
if there is some potential execution of c
where x is read before it is (possibly) written.
Implicitly, every variable is read at the end of c.

Examples: Is x initially dead or live?

$x ::= N\ 0$	☹
$y ::= V\ x;\ y ::= N\ 0;\ x ::= N\ 0$	☺
$WHILE\ b\ DO\ y ::= V\ x;\ x ::= N\ 1$	☺

At the end of a command, we may be interested in the value of *only some of the variables*, e.g. *only the global variables* at the end of a procedure.

Then we say that x is live before c relative to the set of variables X.

Liveness analysis

$L :: com \Rightarrow name\ set \Rightarrow name\ set$

$$L\ c\ X\ =\ \text{live before } c \text{ relative to } X$$

$$
\begin{array}{lcl}
L\ SKIP\ X & = & X \\
L\ (x ::= a)\ X & = & X - \{x\} \cup vars\ a \\
L\ (c_1;\ c_2)\ X & = & (L\ c_1 \circ L\ c_2)\ X \\
L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X & = & \\
\quad vars\ b \cup L\ c_1\ X \cup L\ c_2\ X & & \\
L\ (WHILE\ b\ DO\ c)\ X & = & vars\ b \cup X \cup L\ c\ X
\end{array}
$$

Examples:

$L\ (1 ::= V\ 2;\ 0 ::= Plus\ (V\ 1)\ (V\ 2))\ \{0\}\ =\ \{2\}$

$L\ (WHILE\ Less\ (V\ 0)\ (V\ 0)\ DO\ 1 ::= V\ 2)\ \{0\} = \{0,2\}$

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow T\ set \Rightarrow T\ set$
is called gen/kill analysis
if there are functions gen and $kill$ such that

$$A\ c\ X = X - kill\ c \cup gen\ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$X_1 \subseteq X_2 \Longrightarrow A\ c\ X_1 \subseteq A\ c\ X_2$$
$$A\ c\ (X_1 \cap X_2) = A\ c\ X_1 \cap A\ c\ X_2$$

All the "standard" data-flow analyses are gen/kill.
In particular liveness analysis.

Liveness via gen/kill

$kill :: com \Rightarrow name\ set$

$$
\begin{array}{lcl}
kill\ SKIP & = & \emptyset \\
kill\ (x ::= a) & = & \{x\} \\
kill\ (c_1;\ c_2) & = & kill\ c_1 \cup kill\ c_2 \\
kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = & kill\ c_1 \cap kill\ c_2 \\
kill\ (WHILE\ b\ DO\ c) & = & \emptyset
\end{array}
$$

$gen :: com \Rightarrow name\ set$

$$
\begin{aligned}
&gen\ SKIP = \emptyset \\
&gen\ (x ::= a) = vars\ a \\
&gen\ (c_1;\ c_2) = gen\ c_1 \cup (gen\ c_2 - kill\ c_1) \\
&gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = \\
&\quad vars\ b \cup gen\ c_1 \cup gen\ c_2 \\
&gen\ (WHILE\ b\ DO\ c) = vars\ b \cup gen\ c
\end{aligned}
$$

$$L\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$$

Proof by induction on c.

An easy but important consequence for later:

$$L\ c\ (L\ w\ X) \subseteq L\ w\ X \text{ where } w = \mathit{WHILE}\ b\ \mathit{DO}\ c$$

Do not try to prove this from the original definition of L!

Definite assignment via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$
\begin{array}{lcl}
\mathit{gen}\ \mathit{SKIP} & = & \emptyset \\
\mathit{gen}\ (x ::= a) & = & \{x\} \\
\mathit{gen}\ (c_1;\ c_2) & = & \mathit{gen}\ c_1 \cup \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2) & = & \mathit{gen}\ c_1 \cap \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c) & = & \emptyset
\end{array}
$$

$$\mathit{kill}\ c\ =\ \emptyset$$

12 Live Variable Analysis

Soundness of L

Dead Variable Elimination

Comparisons

$(.,.) \Rightarrow .$ and L should roughly be related like this:

The value of the final state on X
only depends on
the value of the initial state on L c X.

Put differently:

If two initial states agree on L c X
then the corresponding final states agree on X.

Equality on

An abbreviation:

$$f = g \ on \ X \quad \equiv \quad \forall\, x \in X.\ f\, x = g\ x$$

Two easy theorems (in theory $Vars$):

$$s_1 = s_2 \ on \ vars \ a \Longrightarrow aval \ a \ s_1 = aval \ a \ s_2$$
$$s_1 = s_2 \ on \ vars \ b \Longrightarrow bval \ b \ s_1 = bval \ b \ s_2$$

Soundness of L

If $(c,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
then $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof by rule induction

12 Live Variable Analysis

Bury all assignments to dead variables:

$bury :: com \Rightarrow name\ set \Rightarrow com$

$$
\begin{array}{lcl}
bury\ SKIP\ X & = & SKIP \\
bury\ (x ::= a)\ X & = & \textit{if}\ x \in X\ \textit{then}\ x ::= a\ \textit{else}\ SKIP \\
bury\ (c_1;\ c_2)\ X & = & bury\ c_1\ (L\ c_2\ X);\ bury\ c_2\ X
\end{array}
$$

$$
bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ = \\
\quad IF\ b\ THEN\ bury\ c_1\ X\ ELSE\ bury\ c_2\ X
$$

$$
bury\ (WHILE\ b\ DO\ c)\ X\ = \\
\quad WHILE\ b\ DO\ bury\ c\ (vars\ b \cup X \cup L\ c\ X)
$$

Soundness of $bury$

$$(bury\ c\ UNIV,\ s) \Rightarrow s' \quad \longleftrightarrow \quad (c,\ s) \Rightarrow s'$$

where $UNIV$ is the set of all variables.

The two directions need to be proved separately.

$$(c,\ s) \Rightarrow s' \Longrightarrow (bury\ c\ UNIV,\ s) \Rightarrow s'$$

Follows from generalized statement:

If $(c,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
then $\exists\, t'.\ (bury\ c\ X,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof by rule induction, like for soundness of L.

$$(bury\ c\ UNIV,\ s) \Rightarrow s' \Longrightarrow (c,\ s) \Rightarrow s'$$

Follows from generalized statement:

If $(bury\ c\ X,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
then $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof very similar to other direction, but needs inversion lemmas for $bury$ for every kind of command, e.g.

$$\begin{aligned}
&(bc_1;\ bc_2 = bury\ c\ X) = \\
&(\exists\, c_1\ c_2. \\
&\quad c = c_1;\ c_2 \wedge \\
&\quad bc_2 = bury\ c_2\ X \wedge bc_1 = bury\ c_1\ (L\ c_2\ X))
\end{aligned}$$

12 Live Variable Analysis

Soundness of L

Dead Variable Elimination

Comparisons

Comparison of analyses

- Definite assignment analysis is a forward must analysis:
  - it analyses the executions starting from some point,
  - variables *must* be assigned (on every program path) before they are used.

- Live variable analysis is a backward may analysis:
  - it analyses the executions ending in some point,
  - live variables *may* be used (on some program path) before they are assigned.

Comparison of DFA frameworks

Program representation:

- Traditionally (e.g. Aho/Sethi/Ullman), DFA is performed on *control flow graphs* (CFGs).
  Application: optimization of intermediate or low-level code.
- We analyse structured programs.
  Application: source-level program optimization.

Algorithm:

- Gen/kill analyses on arbitrary CFGs may require a finite number of iterations before a (least or greatest) solution is reached.
- Gen/kill analyes of structured programs do not require iterations.

The aim:

Ensure that programs protect private data
like passwords, bank details, or medical records.
There should be no information flow
from private data into public channels.

This is know as information flow control.

Language based security is an approach to information flow control where data flow analysis is used to determine whether a program is free of illicit information flows.

LBS guarantees confidentiality by program analysis, not by cryptography.

These analyses are often expressed as type systems.

Security levels

- Program variables have *security/confidentiality levels*.
- Security levels are partially ordered:
  $l < l'$ means that l is less confidential than l'.
- We identify security levels with nat.
  Level 0 is public.
- Other popular choices for security levels:
  - only two levels, *high* and *low*.
  - the set of security levels is a lattice.

Two kinds of illicit flows

Explicit: `low := high`

Implicit:
```
if high1 = high2 then low := 1
else low := 0
```

Noninterference

High variables do not interfere with low ones.

A variation of confidential input does not cause a variation of public output.

Program c guarantees noninterference iff for all s_1, s_2:

If s_1 *and* s_2 *agree on low variables (but may differ on high variables!), then the states resulting from executing* (c, s_1) *and* (c, s_2) *must also agree on low variables.*

13 Information Flow Control

Security levels:

types $level = nat$

Every variable has a security level:

$sec :: name \Rightarrow level$

No definition is needed. Except for examples.
Hence we define (arbitrarily)

$sec\ n = n$

The security level of an expression is the maximal security level of any of its variables.

$sec :: aexp \Rightarrow level$

$$sec\ (N\ n) = 0$$
$$sec\ (V\ x) = sec\ x$$
$$sec\ (Plus\ a_1\ a_2) = max\ (sec\ a_1)\ (sec\ a_2)$$

$sec :: bexp \Rightarrow level$

$$sec\ (B\ bv) = 0$$
$$sec\ (Not\ b) = sec\ b$$
$$sec\ (And\ b_1\ b_2) = max\ (sec\ b_1)\ (sec\ b_2)$$
$$sec\ (Less\ a_1\ a_2) = max\ (sec\ a_1)\ (sec\ a_2)$$

Agreement of states up to a certain level:

$$s_1 = s_2\ (\le\ l) \quad \equiv \quad \forall\, x.\ sec\ x \le l \longrightarrow s_1\ x = s_2\ x$$

$$s_1 = s_2\ (<\ l) \quad \equiv \quad \forall\, x.\ sec\ x < l \longrightarrow s_1\ x = s_2\ x$$

Noninterference for expressions:

$$[\![s_1 = s_2\ (\le\ l);\ sec\ a \le l]\!] \Longrightarrow aval\ a\ s_1 = aval\ a\ s_2$$
$$[\![s_1 = s_2\ (\le\ l);\ sec\ b \le l]\!] \Longrightarrow bval\ b\ s_1 = bval\ b\ s_2$$

13 Information Flow Control

Explicit flows are easy. How to check for implicit flows:

Carry the security level of the boolean expressions around that guard the current command.

The well-typedness predicate:

$$l \vdash c$$

Intended meaning:
"In the context of boolean expressions of level $\leq l$, command c is well-typed."

Hence:
"Assignments to variables of level $< l$ are forbidden."

Well-typed or not?

$$\mathit{0} \vdash \mathit{IF}\ \mathit{Less}\ (\mathit{V}\ \mathit{0})\ (\mathit{V}\ \mathit{1})\ \mathit{THEN}\ \mathit{1} ::= \mathit{N}\ \mathit{0}\ \mathit{ELSE}\ \mathit{SKIP}$$

$$\mathit{1} \vdash \mathit{IF}\ \mathit{Less}\ (\mathit{V}\ \mathit{0})\ (\mathit{V}\ \mathit{1})\ \mathit{THEN}\ \mathit{1} ::= \mathit{N}\ \mathit{0}\ \mathit{ELSE}\ \mathit{SKIP}$$

$$\mathit{2} \vdash \mathit{IF}\ \mathit{Less}\ (\mathit{V}\ \mathit{0})\ (\mathit{V}\ \mathit{1})\ \mathit{THEN}\ \mathit{1} ::= \mathit{N}\ \mathit{0}\ \mathit{ELSE}\ \mathit{SKIP}$$

The type system

$$l \vdash SKIP$$

$$\frac{sec\ a \leq sec\ x \qquad l \leq sec\ x}{l \vdash x ::= a}$$

$$\frac{l \vdash c_1 \qquad l \vdash c_2}{l \vdash c_1;\ c_2}$$

$$\frac{max\ (sec\ b)\ l \vdash c_1 \qquad max\ (sec\ b)\ l \vdash c_2}{l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2}$$

$$\frac{max\ (sec\ b)\ l \vdash c}{l \vdash WHILE\ b\ DO\ c}$$

Remark:

$l \vdash c$ is syntax-directed and executable.

Anti-monotonicity

$$\frac{l \vdash c \qquad l' \leq l}{l' \vdash c}$$

Proof by ... as usual.

This is often called a subsumption rule
because it says that larger levels subsume smaller ones.

Confinement

If $l \vdash c$ then c cannot modify variables of level $< l$:

$$\frac{(c,\ s) \Rightarrow t \qquad l \vdash c}{s = t\ (< l)}$$

The effect of c is *confined* to variables of level $\geq l$.

Proof by ... as usual.

Noninterference

$$\frac{(c,\ s) \Rightarrow s' \qquad (c,\ t) \Rightarrow t' \qquad \mathit{0} \vdash c \qquad s = t\ (\leq\ l)}{s' = t'\ (\leq\ l)}$$

Proof by . . . as usual.

13 Information Flow Control

The $l \vdash c$ system is intuitive and executable

- but in the literature a more elegant formulation is dominant
- which does not need max
- and works for arbitrary partial orders.

This alternative system $l \vdash' c$ has an explicit subsumption rule

$$\frac{l \vdash' c \qquad l' \leq l}{l' \vdash' c}$$

together with one rule per construct:

$$l \vdash' SKIP$$

$$\frac{sec\ a \leq sec\ x \qquad l \leq sec\ x}{l \vdash' x ::= a}$$

$$\frac{l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' c_1;\ c_2}$$

$$\frac{sec\ b \leq l \qquad l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' IF\ b\ THEN\ c_1\ ELSE\ c_2}$$

$$\frac{sec\ b \leq l \qquad l \vdash' c}{l \vdash' WHILE\ b\ DO\ c}$$

- The subsumption-based system $\vdash'$
  is neither syntax-directed nor directly executable.
- One needs to guess
  where to use a subsumption rule in the derivation.

Equivalence of $\vdash$ and $\vdash'$

$$l \vdash c \Longrightarrow l \vdash' c$$

Proof by induction.
Use subsumption directly below IF and $WHILE$.

$$l \vdash' c \Longrightarrow l \vdash c$$

Proof by induction. Subsumption already a lemma for $\vdash$.

- Systems $l \vdash c$ and $l \vdash' c$ are *top-down*:
  level l comes from the context
  and is checked at $::=$ commands.
- System $\vdash c : l$ is *bottom-up*:
  l is the minimal level of any variable assigned in c
  and is checked at IF and WHILE commands.

$$\vdash SKIP : l$$

$$\frac{sec\ a \leq sec\ x}{\vdash x ::= a : sec\ x}$$

$$\frac{\vdash c_1 : l_1 \qquad \vdash c_2 : l_2}{\vdash c_1;\ c_2 : min\ l_1\ l_2}$$

$$\frac{sec\ b \leq min\ l_1\ l_2 \qquad \vdash c_1 : l_1 \qquad \vdash c_2 : l_2}{\vdash IF\ b\ THEN\ c_1\ ELSE\ c_2 : min\ l_1\ l_2}$$

$$\frac{sec\ b \leq l \qquad \vdash c : l}{\vdash WHILE\ b\ DO\ c : l}$$

Equivalence of $\vdash\ :$ and $\vdash'$

$$\vdash\ c : l \Longrightarrow l \vdash' c$$

Proof by induction.

$$l \vdash' c \Longrightarrow\ \vdash\ c : l$$

Nitpick says: $\mathit{0} \vdash' \mathit{1} ::= N\ \mathit{1}$ but not $\vdash\ \mathit{1} ::= N\ \mathit{1} : \mathit{0}$

$$l \vdash' c \Longrightarrow \exists\, l' \geq l.\ \vdash\ c : l'$$

Proof by induction.

13 Information Flow Control

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond

Does noninterference really guarantee absence of information flow?

$$\frac{(c,\ s) \Rightarrow s' \qquad (c,\ t) \Rightarrow t' \qquad \mathit{0} \vdash c \qquad s = t\ (\leq l)}{s' = t'\ (\leq l)}$$

Beware of covert channels!

$$\mathit{0} \vdash \mathit{WHILE\ Less\ (V\ 1)\ (N\ 1)\ DO\ SKIP}$$

A drastic solution:

WHILE-conditions must not depend on confidential data.

New typing rule:

$$\frac{sec\ b = 0 \qquad 0 \vdash c}{0 \vdash WHILE\ b\ DO\ c}$$

Now provable:

$$\frac{(c,\ s) \Rightarrow s' \qquad 0 \vdash c \qquad s = t\ (\leq l)}{\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ (\leq l)}$$

Further extensions

- Time
- Probability
- Quantitative analysis
- More programming language features:
  - exceptions
  - concurrency
  - OO
  - ...

Literature

The inventors of security type systems are Volpano and Smith.

For an excellent survey see

> Sabelfeld and Myers. *Language-Based Information-Flow Security.* 2003.

Part VI

Hoare Logic

14 Partial Correctness

15 Verification Conditions

16 Totale Correctness

We have proved functional programs correct
(e.g. a compiler).

We have proved properties of imperative languages
(e.g. type safety).

But how do we prove properties of imperative programs?

An example program:

$\mathit{0} ::= N\ \mathit{0};\ \mathit{1} ::= N\ \mathit{0};\ w\ n$

where

$$
\begin{array}{l}
w\ n \equiv \\
\mathit{WHILE}\ \mathit{Less}\ (V\ \mathit{1})\ (N\ n) \\
\mathit{DO}\ (\mathit{1} ::= \mathit{Plus}\ (V\ \mathit{1})\ (N\ \mathit{1}); \\
\qquad \mathit{0} ::= \mathit{Plus}\ (V\ \mathit{0})\ (V\ \mathit{1}))
\end{array}
$$

At the end of the execution,
variable $\mathit{0}$ should contain the sum $\mathit{1} + \ldots + n$.

A proof via operational semantics

Theorem:

$$(0 ::= N\ 0;\ 1 ::= N\ 0;\ w\ n,\ s) \Rightarrow t \Longrightarrow$$
$$t\ 0 = \sum\{1..n\}$$

Required Lemma:

$$(w\ n,\ s) \Rightarrow t \Longrightarrow$$
$$t\ 0 = s\ 0 + \sum\{s\ 1 + 1..n\}$$

Proved by induction.

Hoare Logic provides a *structured* approach for reasoning about properties of states during program execution:

- Rules of Hoare Logic (almost) syntax directed
- Automates reasoning about program execution
- No explicit induction

But no free lunch:

- Must prove implications between predicates on states
- Needs invariants.

This is the standard approach.

Formulas are syntactic objects.

Everything is very concrete and simple.

But complex to formalize.

Hence we soon move to a semantic view of formulas.

Reason for introduction of syntactic approach: didactic

For now, we work with a (syntactically) simplified version of IMP.

Hoare Logic reasons about Hoare triples $\{P\}\ c\ \{Q\}$ where

- P and Q are *syntactic formulas* involving program variables
- P is the precondition, Q is the postcondition
- $\{P\}\ c\ \{Q\}$ means that if P is true at the start of the execution, Q is true at the end of the execution — if the execution terminates! (partial correctness)

Informal example:

$$\{x = 41\}\ x := x + 1\ \{x = 42\}$$

Terminology: P and Q are called assertions.

Examples

$$\{x = 5\} \quad ? \quad \{x = 10\}$$

$$\{True\} \quad ? \quad \{x = 10\}$$

$$\{x = y\} \quad ? \quad \{x \neq y\}$$

Boundary cases:

$$\{True\} \quad ? \quad \{True\}$$

$$\{True\} \quad ? \quad \{False\}$$

$$\{False\} \quad ? \quad \{Q\}$$

The rules of Hoare Logic

$$\{P\}\ SKIP\ \{P\}$$

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

Notation: $Q[a/x]$ means "Q with a substituted for x".

Examples:

$$\begin{array}{lll} \{\qquad\} & x := 5 & \{x = 5\} \\ \{\qquad\} & x := x{+}5 & \{x = 5\} \\ \{\qquad\} & x := 2{*}(x{+}5) & \{x > 20\} \end{array}$$

Intuitive explanation of backward-looking rule:

$$\{Q[a]\}\ x := a\ \{Q[x]\}$$

Afterwards we can replace all occurrences of a in Q by x.

The assignment axiom allows us
to compute the precondition from the postcondition.

There is a version to compute the postcondition from the precondition, but it is more complicated. (Exercise!)

More rules of Hoare Logic

$$\frac{\{P_1\}\ c_1\ \{P_2\} \qquad \{P_2\}\ c_2\ \{P_3\}}{\{P_1\}\ c_1;c_2\ \{P_3\}}$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg\ b\}\ c_2\ \{Q\}}{\{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ WHILE\ b\ DO\ c\ \{P \wedge \neg\ b\}}$$

In the While-rule, P is called an invariant because it is preserved across executions of the loop body.

The consequence rule

So far, the rules were syntax-directed. Now we add

$$\frac{P' \longrightarrow P \qquad \{P\}\ c\ \{Q\} \qquad Q \longrightarrow Q'}{\{P'\}\ c\ \{Q'\}}$$

Preconditions can be strengthened,
postconditions can be weakened.

Two derived rules

Problem with assignment and While-rule:
special form of pre and postcondition.
Better: combine with consequence rule.

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

$$\frac{\{P \wedge b\}\ c\ \{P\} \qquad P \wedge \neg\, b \longrightarrow Q}{\{P\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{Q\}}$$

Example

$$\{True\}$$

$$x := 0;\ y := 0;$$
$$WHILE\ y < n\ DO\ (y := y+1;\ x := x+y)$$

$$\{x = \sum\{1..n\}\}$$

Example proof exhibits key properties of Hoare logic:

- Choice of rules is syntax-directed and hence automatic.
- Proof of ";" proceeds from right to left.
- Proofs require only invariants and arithmetic reasoning.

Assertions are predicates on states

$$assn = state \Rightarrow bool$$

Alternative view: *sets of states*

Semantic approach simplifies meta-theory, our main objective.

Validity

$$\models \{P\}\ c\ \{Q\}$$

$$\longleftrightarrow$$

$$\forall\, s\ t.\ (c,\ s) \Rightarrow t \longrightarrow P\ s \longrightarrow Q\ t$$

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

In contrast:

$$\vdash \{P\}\ c\ \{Q\}$$

"$\{P\}\ c\ \{Q\}$ is provable/derivable"

Provability

$$\vdash \{P\}\ SKIP\ \{P\}$$

$$\vdash \{\lambda s.\ Q\ (s[a/x])\}\ x ::= a\ \{Q\}$$

where $s[a/x] \equiv s(x := aval\ a\ s)$

Example: $\{5 = 5\}\ x := 5\ \{x = 5\}$ in semantic terms:

$$\vdash \{P\}\ 0 ::= N\ 5\ \{\lambda s.\ s\ 0 = 5\}$$

where $P = (\lambda s.\ (s[N\ 5/0])\ 0 = 5) = (\lambda s.\ 5 = 5)$

$$\frac{\vdash \{P\}\ c_1\ \{Q\} \qquad \vdash \{Q\}\ c_2\ \{R\}}{\vdash \{P\}\ c_1;\ c_2\ \{R\}}$$

$$\frac{\begin{array}{c}\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c_1\ \{Q\} \\ \vdash \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}\ c_2\ \{Q\}\end{array}}{\vdash \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}}$$

$$\frac{\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c\ \{P\}}{\vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}}$$

$$\frac{\forall s.\ P'\ s \longrightarrow P\ s \quad \vdash \{P\}\ c\ \{Q\} \quad \forall s.\ Q\ s \longrightarrow Q'\ s}{\vdash \{P'\}\ c\ \{Q'\}}$$

Hoare_Examples.thy

Soundness

Everything that is provable is valid:

$$\vdash \{P\}\ c\ \{Q\} \Longrightarrow \models \{P\}\ c\ \{Q\}$$

Proof by induction, with a nested induction in the While-case.

Towards completeness: $\models \implies \vdash$

Weakest preconditions

The weakest precondition
of command c w.r.t. postcondition Q:

$$wp\ c\ Q = (\lambda s.\ \forall\, t.\ (c,\ s) \Rightarrow\ t \longrightarrow\ Q\ t)$$

The set of states that lead (via c) into Q.

A foundational semantic notion, not merely for the completeness proof.

Nice and easy properties of wp

$$wp\ SKIP\ Q = Q$$

$$wp\ (x ::= a)\ Q = (\lambda s.\ Q\ (s[a/x]))$$

$$wp\ (c_1;\ c_2)\ Q = wp\ c_1\ (wp\ c_2\ Q)$$

$$\begin{aligned} & wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q = \\ & (\lambda s.\ (bval\ b\ s \longrightarrow wp\ c_1\ Q\ s) \wedge \\ & \qquad (\neg\ bval\ b\ s \longrightarrow wp\ c_2\ Q\ s)) \end{aligned}$$

$$\neg\ bval\ b\ s \Longrightarrow wp\ (WHILE\ b\ DO\ c)\ Q\ s = Q\ s$$

$$\begin{aligned} & bval\ b\ s \Longrightarrow \\ & wp\ (WHILE\ b\ DO\ c)\ Q\ s = \\ & wp\ (c;\ WHILE\ b\ DO\ c)\ Q\ s \end{aligned}$$

Completeness

$$\models \{P\}\ c\ \{Q\} \Longrightarrow \vdash \{P\}\ c\ \{Q\}$$

Follows easily if we can prove

$$\vdash \{wp\ c\ Q\}\ c\ \{Q\}$$

Proof by induction on c, for arbitary Q.

Proving program properties by Hoare logic ($\vdash$)
is just as powerful as by operational semantics ($\models$).

WARNING

Most texts that discuss completeness of Hoare logic state or prove that Hoare logic is only "relatively complete" but not complete.

Reason: the standard notion of completeness assumes some abstract mathematical notion of $\models$.

Our notion of $\models$ is defined within the same (limited) proof system (for HOL) as $\vdash$.

Idea:

Reduce provability in Hoare logic to provability in the assertion language: automate the Hoare logic part of the problem.

More precisely:

Generate an assertion C, the verification condition, from $\{P\}\ c\ \{Q\}$ such that

$$\vdash \{P\}\ c\ \{Q\} \quad \textit{iff} \quad C \textit{ is provable.}$$

Method:

Simulate syntax-directed application of Hoare logic rules. Collect all assertion language side conditions.

A problem: loop invariants

Where do they come from?

A trivial solution:

Let the user provide them!

How?

Each loop must be annotated with its invariant!

How to synthesize loop invariants automatically
is a difficult research problem.

Which we ignore here.

Terminology:

VCG = Verification Condition Generator

All successful verification technology for imperative programs relies on

- VCGs (of one kind or another)
- and powerful (semi-)automatic theorem provers.

The (approx.) plan of attack

1. Introduce annotated commands with loop invariants
2. Define functions for *computing*
   - weakest proconditions: $pre :: com \Rightarrow assn \Rightarrow assn$
   - verification conditions: $vc :: com \Rightarrow assn \Rightarrow assn$
3. Soundness: $vc\ c\ Q \Longrightarrow \vdash \{\ ?\ \}\ c\ \{Q\}$
4. Completeness: if $\vdash \{P\}\ c\ \{Q\}$ then c can be annotated (becoming c') such that $vc\ c'\ Q$.

The details are a bit different . . .

Annotated commands

Like commands . . .

$$\textbf{datatype}\ acom = Askip \mid Aassign\ name\ aexp \mid Asemi\ acom\ acom \mid Aif\ bexp\ acom\ acom \mid Awhile\ bexp\ assn\ acom$$

. . . but with an assertion I in $Awhile\ b\ I\ c$.

Example:

$$Awhile\ (Less\ (V\ 1)\ (N\ 5))$$
$$(\lambda s.\ s\ 1 = 0)$$
$$(Aassign\ 1\ (N\ 1))$$

Weakest precondition

$pre :: acom \Rightarrow assn \Rightarrow assn$

$pre\ Askip\ Q = Q$

$pre\ (Aassign\ x\ a)\ Q = (\lambda s.\ Q\ (s[a/x]))$

$pre\ (Asemi\ c_1\ c_2)\ Q = pre\ c_1\ (pre\ c_2\ Q)$

$$pre\ (Aif\ b\ c_1\ c_2)\ Q =$$
$$(\lambda s.\ (bval\ b\ s \longrightarrow pre\ c_1\ Q\ s) \wedge$$
$$(\neg\ bval\ b\ s \longrightarrow pre\ c_2\ Q\ s))$$

$pre\ (Awhile\ b\ I\ c)\ Q = I$

Warning

In the presence of loops,
$pre\ c$ may not be the weakest precondition
but may be anything!

Verification condition

$vc :: acom \Rightarrow assn \Rightarrow assn$

$vc\ Askip\ Q = (\lambda s.\ True)$

$vc\ (Aassign\ x\ a)\ Q = (\lambda s.\ True)$

$vc\ (Asemi\ c_1\ c_2)\ Q =$
$(\lambda s.\ vc\ c_1\ (pre\ c_2\ Q)\ s \wedge vc\ c_2\ Q\ s)$

$vc\ (Aif\ b\ c_1\ c_2)\ Q = (\lambda s.\ vc\ c_1\ Q\ s \wedge vc\ c_2\ Q\ s)$

$vc\ (Awhile\ b\ I\ c)\ Q =$
$(\lambda s.\ (I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s) \wedge$
$\quad (I\ s \wedge bval\ b\ s \longrightarrow pre\ c\ I\ s) \wedge vc\ c\ I\ s)$

Verification conditions only arise from loops:

- the invariant must be invariant
- and it must imply the postcondition.

Everything else in the definition of vc is just bureaucracy: collecting assertions and passing them around.

Hoare triples operate on com,
functions pre and vc operate on $acom$.
Therefore we define

$$astrip :: acom \Rightarrow com$$

$$astrip\ Askip = SKIP$$
$$astrip\ (Aassign\ x\ a) = x ::= a$$
$$astrip\ (Asemi\ c_1\ c_2) = astrip\ c_1;\ astrip\ c_2$$
$$astrip\ (Aif\ b\ c_1\ c_2) =$$
$$IF\ b\ THEN\ astrip\ c_1\ ELSE\ astrip\ c_2$$
$$astrip\ (Awhile\ b\ I\ c) = WHILE\ b\ DO\ astrip\ c$$

Soundness of vc & pre w.r.t. $\vdash$

$$\forall s.\ vc\ c\ Q\ s \Longrightarrow\ \vdash \{pre\ c\ Q\}\ astrip\ c\ \{Q\}$$

Proof by induction on c, for arbitrary Q.

Corolllary:

$$(\forall s.\ vc\ c\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre\ c\ Q\ s) \Longrightarrow$$
$$\vdash \{P\}\ astrip\ c\ \{Q\}$$

How to prove some $\vdash \{P\}\ c_0\ \{Q\}$:

- Annotate c_0 yielding c, i.e. $astrip\ c = c_0$.
- Prove Hoare-free premise of corollary.

But is premise provable if $\vdash \{P\}\ c_0\ \{Q\}$ is?

$$(\forall s.\ vc\ c\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre\ c\ Q\ s) \Longrightarrow$$
$$\vdash \{P\}\ astrip\ c\ \{Q\}$$

Why could premise not be provable
although conclusion is?

- Some annotation in c is not invariant.
- vc or pre are wrong
  (e.g. accidentally always produce $False$).

Therefore we prove completeness:
suitable annotations exist such that premise is provable.

Completeness of vc & pre w.r.t. $\vdash$

$$\vdash \{P\}\ c\ \{Q\} \Longrightarrow$$
$$\exists\, c'.\ astrip\ c' = c\ \wedge$$
$$(\forall\, s.\ vc\ c'\ Q\ s) \wedge (\forall\, s.\ P\ s \longrightarrow pre\ c'\ Q\ s)$$

Proof by rule induction. Needs two monotonicity lemmas:

$$[\![\forall\, s.\ P\ s \longrightarrow P'\ s;\ pre\ c\ P\ s]\!] \Longrightarrow pre\ c\ P'\ s$$

$$[\![\forall\, s.\ P\ s \longrightarrow P'\ s;\ vc\ c\ P\ s]\!] \Longrightarrow vc\ c\ P'\ s$$

- Partial Correctness:
  if command terminates, postcondition holds
- Total Correctness:
  command terminates *and* postcondition holds

Total Correctness = Partial Correctness + Termination

Formally:

$$\models_t \{P\}\ c\ \{Q\} \equiv \forall\, s.\ P\ s \longrightarrow (\exists\, t.\ (c,\ s) \Rightarrow t \wedge\ Q\ t)$$

Assumes that semantics is deterministic!

Exercise: Reformulate for nondeterministic language

$\vdash_t$: A proof system for total correctness

Only need to change the While-rule.

Some measure function $state \Rightarrow nat$
must decrease with every loop iteration

$$\frac{\bigwedge n.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge f\ s = n\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}}{\vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}}$$

HoareT.thy

Example

Soundness

$$\vdash_t \{P\}\ c\ \{Q\} \Longrightarrow \models_t \{P\}\ c\ \{Q\}$$

Proof by induction, with a nested induction (on what?) in the While-case.

Completeness

$$\models_t \{P\}\ c\ \{Q\} \Longrightarrow \vdash_t \{P\}\ c\ \{Q\}$$

Follows easily from

$$\vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$$

where

$$wp_t\ c\ Q \equiv \lambda s.\ \exists\, t.\ (c,\ s) \Rightarrow\ t \wedge\ Q\ t.$$

Proof of $\vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$ is by induction on c.

In the $WHILE\ b\ DO\ c$ case, let $f\ s$ (in the $\vdash_t$ rule for While) be the number of iterations that the loop needs if started in state s.

This f depends on b and c and is definable in HOL.

Part VII

Extensions of IMP

17 Procedures and Local Variables

New commands

Declare local variable: $\{VAR\ x;;\ c\}$

Define local procedure: $\{PROC\ p = c;;\ c'\}$

Call procedure: $CALL\ p$

Concrete syntax

$$\begin{array}{lcl} com & ::= & \ldots \text{basic commands} \ldots \\ & | & \{VAR\ name;;\ com\} \\ & | & \{PROC\ name = com;;\ com\} \\ & | & CALL\ name \end{array}$$

Abstract syntax

$$\begin{array}{rcl}\textbf{datatype } com & = & \ldots \text{basic commands} \ldots \\ & | & Var\ name\ com \\ & | & Proc\ name\ com\ com \\ & | & CALL\ name\end{array}$$

Scoping

Static scoping

Name n refers to the textually enclosing declaration of n in the program text.

Dynamic scoping

Name n refers to the most recent declaration of n during execution.

Example

$$
\begin{array}{l}
\{VAR\ 0;;\ 0 ::= N\ 0; \\
\ \{PROC\ 0 = 0 ::= Plus\ (V\ 0)\ (V\ 0);; \\
\ \ \{PROC\ 1 = CALL\ 0;; \\
\ \ \ \{VAR\ 0;;\ 0 ::= N\ 5; \\
\ \ \ \ \{PROC\ 0 = 0 ::= Plus\ (V\ 0)\ (N\ 1);; \\
\ \ \ \ \ CALL\ 1;\ 1 ::= V\ 0\}\}\}\}\}
\end{array}
$$

What is the final value of variable 1 ?

- static scope for VAR and $PROC$
- dynamic scope for VAR and static scope for $PROC$
- dynamic scope for VAR and $PROC$

C does not allow nested procedures,
which simplifies the semantics.

Most functional languages allow nested procedures.

As does Java, via inner classes.

Dynamic scoping is a concept from hell and rarely used.

But its semantics is easy to define
and a good starting point.

Procedure environment

$$penv \;=\; name \Rightarrow com$$

Big-step semantics:

$$pe \vdash (c,\ s) \Rightarrow t$$

where $pe :: penv$.
Rules for basic commands are upgraded by adding $pe \vdash$.
Example:

$$\frac{pe \vdash (c_1,\ s_1) \Rightarrow s_2 \qquad pe \vdash (c_2,\ s_2) \Rightarrow s_3}{pe \vdash (c_1;\ c_2,\ s_1) \Rightarrow s_3}$$

Rules for new commands

$$\frac{pe \vdash (c,\ s) \Rightarrow t}{pe \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)}$$

$$\frac{pe(p := cp) \vdash (c,\ s) \Rightarrow t}{pe \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t}$$

$$\frac{pe \vdash (pe\ p,\ s) \Rightarrow t}{pe \vdash (CALL\ p,\ s) \Rightarrow t}$$

Dynamic scoping because $pe(n)$ and $s(n)$ are the current values of n w.r.t. execution.

17 Procedures and Local Variables

Introduction

Dynamic Scope for VAR and PROC

Dynamic Scope for VAR, Static Scope for PROC

Static Scope for VAR and PROC

The static environment for a procedure p is the procedure environment at the point where p is declared, i.e. the static links to the procedures known at that point.

Recorde the static environment for each procedure together with the procedure body:

$$penv \;=\; name \Rightarrow com \times penv$$

Recursive type synonyms not allowed.
Alternative: organize procedure environment like a stack.

$$penv \;=\; (name \times com)\; list$$

The static environment of p is the $penv$ before $(p, \_)$ was added: pop until $(p, \_)$ is found.

Rules for new commands

$$\frac{pe \vdash (c,\ s) \Rightarrow t}{pe \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)}$$

$$\frac{(p,\ cp) \# pe \vdash (c,\ s) \Rightarrow t}{pe \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t}$$

$$\frac{(p,\ c) \# pe \vdash (c,\ s) \Rightarrow t}{(p,\ c) \# pe \vdash (CALL\ p,\ s) \Rightarrow t}$$

$$\frac{p' \neq p \qquad pe \vdash (CALL\ p,\ s) \Rightarrow t}{(p',\ c) \# pe \vdash (CALL\ p,\ s) \Rightarrow t}$$

17 Procedures and Local Variables

Separate variable names from their storage addresses. The same x can have different addresses at different points in the program.

$$addr \ = \ nat$$

A variable environment associates names with addresses:

$$venv \ = \ name \Rightarrow addr$$

A store associates addresses with values:

$$store \ = \ addr \Rightarrow nat$$

Note: If $s :: store$ and $ve :: venv$ then $s \circ ve :: state$.

The static environment for each procedure p records both

- the procedure environment and
- the variable environment

at the point where p is declared.

The procedure environment is recorded as before (in the stack), the variable environment explicitly:

$$penv \;=\; (name \times venv \times com)\; list$$

Interpretation of $(p,\ ve,\ c)$:
variable x in c refers to address $ve(x)$.

Big-step format

Execution takes place in the context of

- a procedure environment pe
- a variable environment ve
- a free address f

Instead of a state, the semantics transforms a store s:

$$(pe,ve,f) \vdash (c,\ s) \Rightarrow t$$

Execution also modifies the context, but input/output behaviour is captured by the store transformation.

Auxiliary function: $venv\ (pe,\ ve,\ f) = ve$

Rules for basic commands

$$e \vdash (SKIP,\ s) \Rightarrow s$$

$$(pe,\ ve,\ f) \vdash (x ::= a,\ s) \Rightarrow s(ve\ x := aval\ a\ (s \circ ve))$$

$$\frac{e \vdash (c_1,\ s_1) \Rightarrow s_2 \qquad e \vdash (c_2,\ s_2) \Rightarrow s_3}{e \vdash (c_1;\ c_2,\ s_1) \Rightarrow s_3}$$

$$\frac{bval\ b\ (s \circ venv\ e) \qquad e \vdash (c_1,\ s) \Rightarrow t}{e \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

$$\frac{\neg\ bval\ b\ (s \circ venv\ e) \qquad e \vdash (c_2,\ s) \Rightarrow t}{e \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

$$\frac{\neg\ bval\ b\ (s \circ venv\ e)}{e \vdash (WHILE\ b\ DO\ c,\ s) \Rightarrow s}$$

$$\frac{bval\ b\ (s_1 \circ venv\ e) \quad e \vdash (c,\ s_1) \Rightarrow s_2 \qquad e \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3}{e \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3}$$

Rules for new commands

$$\frac{(pe,\ ve(x := f),\ f + 1) \vdash (c,\ s) \Rightarrow t}{(pe,\ ve,\ f) \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)}$$

$$\frac{((p,\ cp,\ ve) \# pe,\ ve,\ f) \vdash (c,\ s) \Rightarrow t}{(pe,\ ve,\ f) \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t}$$

$$\frac{((p,\ c,\ ve) \# pe,\ ve,\ f) \vdash (c,\ s) \Rightarrow t}{((p,\ c,\ ve) \# pe,\ ve',\ f) \vdash (CALL\ p,\ s) \Rightarrow t}$$

$$\frac{p' \neq p \qquad (pe,\ ve,\ f) \vdash (CALL\ p,\ s) \Rightarrow t}{((p',\ c,\ ve') \# pe,\ ve,\ f) \vdash (CALL\ p,\ s) \Rightarrow t}$$

Motto

Addresses are numbers, too!

We take full advantage of

$$state = nat \Rightarrow nat$$

Arithmetic expressions

$$\textbf{datatype}\ aexp \ = \ N\ nat \ \mid \ Deref\ aexp \ \mid \ Plus\ aexp\ aexp$$

- Syntax: $!\,a \;\equiv\; Deref\ a$
- Pronounced "contents of a"
- Allows terms like $!\,(Plus\ (!\,(N\ 5))\ (N\ 2))$.

Why no variables?

Numbers are addresses are variables.

Instead of $V\ 1$ we now write $!\ (N\ 1)$.

C has variables, but you can obtain their address.

We work directly with addresses.

$aval$ and $bval$

$aval :: aexp \Rightarrow state \Rightarrow nat$

$$
\begin{array}{lcl}
aval\ (N\ n)\ s & = & n \\
aval\ (!\ a)\ s & = & s\ (aval\ a\ s) \\
aval\ (Plus\ a_1\ a_2)\ s & = & aval\ a_1\ s + aval\ a_2\ s
\end{array}
$$

Function $bval$ remains unchanged.

Assignment

$$aexp ::= aexp$$

Left-hand side is address, righ-hand side is value.

Memory allocation

A new command:

$$New\ aexp\ aexp$$

$New\ a\ k$ allocates a storage block of size k and stores the start address at address a.

Why not make $New\ k$ an $aexp$ that returns the start address as its value?

Big-step semantics

$$(com,\ state,\ nat) \Rightarrow (state,\ nat)$$

- The nat-component is the first free address.
- Everything beyond that address is free, too.
- This free pointer increases monotonically.
- There is no garbage collection.
- This is a very concrete storage allocation policy.
- More abstract nondeterministic models are possible but sacrifice executability.

In Isabelle, tuples are nested pairs:

$$(a,\ b,\ c)\ \equiv\ (a,\ (b,\ c))$$

$$\tau_1 \times \tau_2 \times \tau_3\ \equiv\ \tau_1 \times (\tau_2 \times \tau_3)$$

$\Longrightarrow$ big_step is of type

$$com \times (state \times nat) \Rightarrow (state \times nat) \Rightarrow bool$$

Big-step rules

$$(SKIP,\ sn) \Rightarrow sn$$

$$(lhs ::= a,\ s,\ n) \Rightarrow (s(aval\ lhs\ s := aval\ a\ s),\ n)$$

$$(New\ lhs\ a,\ s,\ n) \Rightarrow (s(aval\ lhs\ s := n),\ n + aval\ a\ s)$$

$$\frac{(c_1,\ sn_1) \Rightarrow sn_2 \qquad (c_2,\ sn_2) \Rightarrow sn_3}{(c_1;\ c_2,\ sn_1) \Rightarrow sn_3}$$

Big-step rules

$$\frac{bval\ b\ s \qquad (c_1,\ s,\ n) \Rightarrow tn}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s,\ n) \Rightarrow tn}$$

$$\frac{\neg\ bval\ b\ s \qquad (c_2,\ s,\ n) \Rightarrow tn}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s,\ n) \Rightarrow tn}$$

Big-step rules

$$\frac{\neg\ bval\ b\ s}{(WHILE\ b\ DO\ c,\ s,\ n) \Rightarrow (s,\ n)}$$

$$\frac{bval\ b\ s_1 \quad (c,\ s_1,\ n) \Rightarrow sn_2 \qquad (WHILE\ b\ DO\ c,\ sn_2) \Rightarrow sn_3}{(WHILE\ b\ DO\ c,\ s_1,\ n) \Rightarrow sn_3}$$

How does assignment differ from C?

In C (and most imperative languages),
the lhs and the rhs are evaluated differently:

- on the lhs, a variable represents its address,
- on the rhs, a variable represents its value.

We use ! to achieve the same effect.

Some array and pointer algorithms

Array summation example

Variables:

$!\ (N\ \mathit{0}) =$ address of first element of array

$!\ (N\ \mathit{1}) =$ address of last element of array

$!\ (N\ \mathit{2}) =$ sum, initially $\mathit{0}$

Linked list creation example

Variables:

$! \; (N \; 0) =$ number of elements to be created

$! \; (N \; 1) =$ counter, initially 0

$! \; (N \; 2) =$ head of list, initially 0

$! \; (N \; 3) =$ aux

List element: (list size, next pointer)

Motto

Everything is an object!

Even natural numbers.

Design decisions

- Every language construct is an expression.
- Every expression evaluates to an object reference.

Expressions exp

$Null$	
New	
$V\ string$	Variable access
$exp{\bullet}string$	Field access
$string ::= exp$	Variable assignment
$exp{\bullet}string ::= exp$	Field assignment
$exp{\bullet}string<exp>$	Method call
$exp;\ exp$	
$IF\ bexp\ THEN\ exp\ ELSE\ exp$	

- Why no $SKIP$?
- Why no $WHILE$?
- Why no multiple parameters?

Boolean expressions $bexp$

$$bexp = B\ bool \mid Not\ bexp \mid And\ bexp\ bexp \mid Eq\ exp\ exp$$

A case of mutually recursive data types

datatype $exp = \ldots\ exp \ldots\ bexp \ldots$
and $bexp = \ldots\ bexp \ldots\ exp \ldots$

References, objects, stores

A reference is $null$ or an address (nat):

$$\textbf{datatype}\ ref = null \mid Ref\ nat$$

An object maps field names to references:

$$obj \ = \ string \Rightarrow ref$$

A store maps addresses to objects:

$$store \ = \ nat \Rightarrow obj$$

Environments

A variable environment maps variable names to references:

$$venv \;=\; string \Rightarrow ref$$

A method environment maps method names to bodies:

$$menv \;=\; string \Rightarrow exp$$

Big-step semantics

$$menv \vdash (exp,\ config) \Rightarrow (ref,\ config)$$

where

$$config \ = \ venv \times store \times nat$$

Big-step rules

$$me \vdash (Null,\ c) \Rightarrow (null,\ c)$$

$$me \vdash$$
$$(New,\ ve,\ s,\ n) \Rightarrow (Ref\ n,\ ve,\ s(n := \lambda f.\ null),\ n + 1)$$

$$me \vdash (V\ x,\ ve,\ sn) \Rightarrow (ve\ x,\ ve,\ sn)$$

$$\frac{me \vdash (e,\ c) \Rightarrow (Ref\ a,\ ve',\ s',\ n')}{me \vdash (e{\bullet}f,\ c) \Rightarrow (s'\ a\ f,\ ve',\ s',\ n')}$$

Big-step rules

$$\frac{me \vdash (e,\ c) \Rightarrow (r,\ ve',\ sn')}{me \vdash (x ::=\ e,\ c) \Rightarrow (r,\ ve'(x := r),\ sn')}$$

$$\frac{\begin{array}{c} me \vdash (oe,\ c_1) \Rightarrow (Ref\ a,\ c_2) \\ me \vdash (e,\ c_2) \Rightarrow (r,\ ve_3,\ s_3,\ n_3) \end{array}}{me \vdash (oe{\bullet}f ::=\ e,\ c_1) \Rightarrow (r,\ ve_3,\ s_3(a{,}f := r),\ n_3)}$$

where $f(x{,}y := z) \equiv f(x := (f\ x)(y := z))$

Big-step rules

$$\frac{\begin{array}{c} me \vdash (oe,\ c_1) \Rightarrow (or,\ c_2) \\ me \vdash (pe,\ c_2) \Rightarrow (pr,\ ve_3,\ sn_3) \\ ve = (\lambda x.\ null)(''this'' := or,\ ''param'' := pr) \\ me \vdash (me\ m,\ ve,\ sn_3) \Rightarrow (r,\ ve',\ sn_4) \end{array}}{me \vdash (oe{\bullet}m{<}pe{>},\ c_1) \Rightarrow (r,\ ve_3,\ sn_4)}$$

Big-step rules

$$\frac{me \vdash (e_1,\ c_1) \Rightarrow (r,\ c_2) \qquad me \vdash (e_2,\ c_2) \Rightarrow\ c_3}{me \vdash (e_1;\ e_2,\ c_1) \Rightarrow\ c_3}$$

$$\frac{me \vdash (b,\ c_1) \rightarrow (True,\ c_2) \qquad me \vdash (e_1,\ c_2) \Rightarrow\ c_3}{me \vdash (IF\ b\ THEN\ e_1\ ELSE\ e_2,\ c_1) \Rightarrow\ c_3}$$

$$\frac{me \vdash (b,\ c_1) \rightarrow (False,\ c_2) \qquad me \vdash (e_2,\ c_2) \Rightarrow\ c_3}{me \vdash (IF\ b\ THEN\ e_1\ ELSE\ e_2,\ c_1) \Rightarrow\ c_3}$$

Evaluation of $bexp$

$$menv \vdash (bexp,\ config) \rightarrow (bool,\ config)$$

The rules are the obvious ones.

A case of mutually inductive predicates:

inductive
$big\_step :: menv \Rightarrow exp \times config \Rightarrow ref \times config \Rightarrow bool$
and
$bval :: menv \Rightarrow bexp \times config \Rightarrow bool \times config \Rightarrow bool$

Natural numbers as objects

- 0 is represented by $null$.
- $n{+}1$ is represented by an object with a predecessor field that points to a representation of n.

Successor method:

$$("s" ::= New)\bullet"pred" ::= V\ "this";\ V\ "s"$$

Addition method:

$$\begin{array}{l} IF\ Eq\ (V\ "param")\ Null\ THEN\ V\ "this" \\ ELSE\ V\ "this"\bullet"succ"<Null>\bullet \\ \qquad "add"<V\ "param"\bullet"pred"> \end{array}$$

OO?

Which central OO feature is missing?

Dynamic method binding

In $oe{\bullet}m{<}e{>}$, the name m determines the method, the object has no influence.

Two possible extensions:

- Attach the method body to each object, like the fields.
- Superimpose a class system and attach a class to each object.