Concrete Semantics

—

A Proof Assistant Approach

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

Wintersemester 2012

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:

Proof Assistant (PA)
or
Interactive Theorem Prover (ITP)

Proof Assistants

- You give the structure of the proof
- The PA 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 PAs

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

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 (PA!)

Our PA: 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 PA 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 a PA is leading edge

A growing number of universities offer related course

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. Induction and Simplification

5. Case Study: IMP Expressions

6. Logic and Proof beyond "="

7. Isar: A Language for Structured Proofs

Notation

Implication associates to the right:

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

Similarly for other arrows: $\Rightarrow$, $\longrightarrow$

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

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: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\ldots$

2 Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types $bool$, nat and $list$

Summary

Types

Basic syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	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: $\lambda x.\ plus\ x\ x$

Terms

Basic 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:

! $(if\ \_\ then\ \_\ else\ \_)$ **!**

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`

2 Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types $bool$, nat and $list$

Summary

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")
  (eg $\Longrightarrow$ instead of `==>`, $\forall$ instead of `ALL`)

isabelle jedit

Similar to ProofGeneral but

- based on jedit
- $\Longrightarrow$ easier to install
- $\Longrightarrow$ may be more familiar
- Has advantages and a few disadvantages

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

Types and terms

Interfaces

By example: types $bool$, nat and $list$

Summary

Type $bool$

datatype $bool = True \mid False$

Predefined functions:

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

A *formula* is a term of type $bool$

if-and-only-if: $=$

Type nat

datatype $nat \;=\; 0 \;|\; Suc\ nat$

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

An informal proof

Lemma $add\ m\ 0 = m$

Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).
  We need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:

$$
\begin{array}{lcll}
add\ (Suc\ m)\ 0 & = & Suc\ (add\ m\ 0) & \text{by def. of } add \\
 & = & Suc\ m & \text{by IH}
\end{array}
$$

Type $'a\ list$

Lists of elements of type $'a$

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

Some lists: Nil, $Cons\ 1\ Nil$, $Cons\ 1\ (Cons\ 2\ Nil)$, ...

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

An informal proof

Lemma $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$
Proof by induction on xs.

- Case Nil: $app\ (app\ Nil\ ys)\ zs = app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$ holds by definition of app.
- Case $Cons\ x\ xs$: We assume $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$ (IH), and we need to show
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs =$
  $app\ (Cons\ x\ xs)\ (app\ ys\ zs)$.
  The proof is as follows:
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs$
  $= Cons\ x\ (app\ (app\ xs\ ys)\ zs)$ by definition of app
  $= Cons\ x\ (app\ xs\ (app\ ys\ zs))$ by IH
  $= app\ (Cons\ x\ xs)\ (app\ ys\ zs)$ by definition of app

Large library: HOL/List.thy

Included in Main.

Don't reinvent, reuse!

Predefined: $xs \mathbin{@} ys$ (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

Types and terms

Interfaces

By example: types $bool$, nat and $list$

Summary

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

Proof methods

- *induction* 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

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

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

Preview: Multiple assumptions

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

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$;$ $\approx$ "and"

3 Type and function definitions

Type definitions

Function definitions

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples:

type_synonym $string = char\ list$

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

Type synonyms 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 $case$:

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

Wildcards: _

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy

The *option* type

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

If $'a$ has values $a_1,\ a_2,\ \dots$
then $'a\ option$ has values $None$, $Some\ a_1$, $Some\ a_2$, $\dots$

Typical application:

fun $lookup :: ('a \times\ 'b)\ list \Rightarrow\ 'a \Rightarrow\ 'b\ option$ **where**
$lookup\ []\ x = None\ |$
$lookup\ ((a,b)\ \#\ ps)\ x =$
$\quad (if\ a = x\ then\ Some\ b\ else\ lookup\ ps\ x)$

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 $f\, x_1\ \ldots\ x_n\ =\ \ldots$

The danger of nontermination

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**

$$
\begin{aligned}
&ack\ 0\ n = Suc\ n \mid \\
&ack\ (Suc\ m)\ 0 = ack\ m\ (Suc\ 0) \mid \\
&ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)
\end{aligned}
$$

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

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

primrec

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

The essence of primitive recursion:

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

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

4 Induction and Simplification

Induction

Simplification

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

Induction_Demo.thy

Generalisation

Generalisation

- Replace constants by variables
- Generalize free variables
  - by $arbitrary$ in induction proof
  - (or by universal quantifier in formula)

So far, all proofs were by structural induction because all functions were 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`:

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

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

$$(induction\ a_1\ \ldots\ a_n\ rule\colon f.induct)$$

Heuristic:

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

`Induction_Demo.thy`

Computation Induction

4 Induction and Simplification

Induction

Simplification

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:

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

Rewriting:

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

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$(simp\ add{:}\ eq_1\ \ldots\ eq_n)$

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:

- $(simp \ldots del{:} \ldots)$ 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:

$$(simp\ add\colon f\_def \ldots)$$

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: $(\mathit{simp}\ \mathit{split}\colon \mathit{nat.split})$
Or auto. Similar for any datatype t: $\mathit{t.split}$

Simp_Demo.thy

This section introduces

arithmetic and boolean expressions

of our imperative language IMP.

IMP *commands* are introduced later.

5 Case Study: IMP Expressions

Arithmetic Expressions

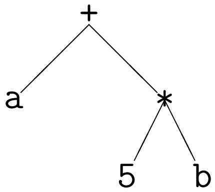
Boolean Expressions

Stack Machine 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 strings, values are integers:

type_synonym $vname = string$
datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

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

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:

type_synonym $val = int$
type_synonym $state = vname \Rightarrow val$

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.

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

How to write down a state

Some states:

- $\lambda x.\ \mathit{0}$
- $(\lambda x.\ \mathit{0})(''a'' := \mathit{3})$
- $((\lambda x.\ \mathit{0})(''a'' := \mathit{5}))(''x'' := \mathit{3})$

Nicer notation:

$$<''a'' := \mathit{5},\ ''x'' := \mathit{3},\ ''y'' := \mathit{7}>$$

Maps everything to $\mathit{0}$, but $''a''$ to $\mathit{5}$, $''x''$ to $\mathit{3}$, etc.

AExp.thy

5 Case Study: IMP Expressions

BExp.thy

5 Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine 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.

Syntax (in decreasing precedence):

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

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 precedence as $\longrightarrow$)

Variable binding convention:

$$\forall\, x\ y.\ P\ x\ y \ \equiv\ \forall\, x.\ \forall\, y.\ P\ x\ y$$

Similarly for $\exists$ and λ.

Warning

Quantifiers have low precedence
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:

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

Sets over type $'a$

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

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

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

Set comprehension

- $\{x.\ P\}$ where x is a variable
- But not $\{t.\ P\}$ where t is a proper term
- Instead: $\{t\ |x\ y\ z.\ P\}$
  is short for $\{v.\ \exists\, x\ y\ z.\ v = t \wedge P\}$
  where x, y, z are the variables in t.

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

fastforce

- 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(*proof-method*)

$\approx$

apply(*proof-method*)
done

Auto_Proof_Demo.thy

6 Logic and Proof beyond "="

Logical Formulas

Proof Automation

Single Step Proofs

Inductive Definitions

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

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: $\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

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

$2.\ \ldots \Longrightarrow B$

The general case: applying rule $\llbracket A_1; \ldots ; A_n \rrbracket \Longrightarrow A$ 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

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

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**($blast\ intro\text{: } trans$)

Can greatly increase the search space!

Forward proof: OF

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

$$\rightsquigarrow$$

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

From now on: $\wr$ mostly suppressed on slides

Single_Step_Demo.thy

$\Longrightarrow$ versus $\longrightarrow$

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

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

Phrase theorems like this $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$
not like this $A_1 \wedge \ldots \wedge A_n \longrightarrow A$

6 Logic and Proof beyond "="

Logical Formulas

Proof Automation

Single Step Proofs

Inductive Definitions

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
$ev\ 0\quad |$
$ev\ n \Longrightarrow ev\ (n + 2)$

An easy proof: $ev\ 4$

$$ev\ 0 \Longrightarrow ev\ 2 \Longrightarrow ev\ 4$$

Consider

fun $even :: nat \Rightarrow bool$ **where**
$even\ 0 = True\ |$
$even\ (Suc\ 0) = False\ |$
$even\ (Suc\ (Suc\ n)) = even\ n$

A trickier proof: $ev\ m \Longrightarrow even\ m$

By induction on the *structure* of the derivation of $ev\ m$

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow even\ m = True$
- rule $ev\ n \Longrightarrow ev\ (n{+}2)$
  $\Longrightarrow m = n{+}2$ and $even\ n$ (IH)
  $\Longrightarrow even\ m = even\ (n{+}2) = even\ n = True$

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`:

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n.\ [\![\ ev\ n;\ P\ n\ ]\!] \Longrightarrow P(n{+}2)}{P\ n}$$

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

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

that P is preserved:

$$[\![\ I\ a_1;\ P\ a_1;\ \ldots\ ;\ I\ a_n;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

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

Inductive_Demo.thy

Inductively defined sets

inductive_set $I :: \tau\ set$ **where**

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

$\vdots$

Difference to **inductive**:

- arguments of I are tupled, not curried
- I can later be used with set theoretic operators, eg $I \cup \ldots$

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** $simp$
  ⋮
  have $formula_n$ **by** $blast$
  show $formula_{n+1}$ **by** $\ldots$
qed

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

Isar core syntax

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

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

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 $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof default proof: assume *surj*, show *False*
  assume a: $surj\ f$
  from a **have** b: $\forall\ A.\ \exists\ a.\ A = f\ a$
    by$(simp\ add\colon surj\_def)$
  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
  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

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

Contradiction

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

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

$\longleftrightarrow$

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

$\forall$ 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*
⋮ $\quad x$ fixed local variable

Works for one or more x

obtain example

lemma $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof
  assume $surj\ f$
  hence $\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**$(auto\ simp\colon surj\_def)$
  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

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

Isar_Demo.thy

Exercise

7 Isar: A Language for Structured Proofs

Example: pattern matching

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

$?thesis$

show *formula* (is $?thesis$)
proof -
$\vdots$
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 $(\exists\, ys.\ xs = ys\ @\ rev\ ys) \vee$
$(\exists\, ys\ a.\ xs = ys\ @\ a \# rev\ ys)$
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 ...
moreover
have P_2 ...
moreover
$\vdots$
moreover
have P_n ...
ultimately
have P ...

$\approx$

have lab_1: P_1 ...
have lab_2: P_2 ...
$\vdots$
have lab_n: P_n ...
from lab_1 lab_2 ...
have P ...

With names

Raw proof blocks

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

proves $[\![\ A_1; \ \ldots\ ; A_m\ ]\!] \Longrightarrow B$
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):

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

How to prove each subgoal:

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

Separated by **next**

Isar_Induction_Demo.thy

Case distinction

Datatype case distinction

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

proof $(cases$ "$term$"$)$
  case $(C_1\ x_1\ \ldots\ x_k)$
  $\ldots\ x_j\ \ldots$
next
$\vdots$
qed

where **case** $(C_i\ x_1\ \ldots\ x_k)$ $\equiv$

fix $x_1\ \ldots\ x_k$
assume $\underbrace{C_i:}_{\text{label}}\ \underbrace{term = (C_i\ x_1\ \ldots\ x_k)}_{\text{formula}}$

Isar_Induction_Demo.thy

Structural induction for nat

Structural induction for nat

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

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$
proof $(induction\ n)$

case *0*	$\equiv$	**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

Named assumptions

In a proof of

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by structural induction:
In the context of

case C

we have

$C.IH$	the induction hypotheses
$C.prems$	the premises A_i
C	$C.IH + C.prems$

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_Induction_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 *(induction 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 $x_1\ \ldots\ x_k$.

Named assumptions

In a proof of

$$I \ldots \implies A_1 \implies \ldots \implies A_n \implies B$$

by rule induction on $I \ldots$:
In the context of

case R

we have

$R.IH$	the induction hypotheses
$R.hyps$	the assumptions of rule R
$R.prems$	the premises A_i
R	$R.IH + R.hyps + R.prems$

7 Isar: A Language for Structured Proofs

Rule inversion

inductive $ev :: nat \Rightarrow bool$ **where**
$ev0$: $ev\ 0$ |
$evSS$: $ev\ n \Longrightarrow ev(Suc(Suc\ n))$

What can we deduce from $ev\ n$?
That it was proved by either $ev0$ or $evSS$!

$$ev\ n \Longrightarrow n = 0 \vee (\exists k.\ n = Suc\ (Suc\ k) \wedge ev\ k)$$

Rule inversion = case distinction over rules

Isar_Induction_Demo.thy

Rule inversion

Rule inversion template

from $`ev\ n`$ **have** P
proof $cases$
  case $ev0$ $\qquad n = 0$
  $\vdots$
  show $?thesis$ ...
next
  case $(evSS\ k)$ $\qquad n = Suc\ (Suc\ k),\ ev\ k$
  $\vdots$
  show $?thesis$ ...
qed

Impossible cases disappear automatically

Part II

IMP: A Simple Imperative Language

8. IMP

9. Compiler

10. A Typed Version of IMP

Terminology

Statement: declaration of fact or claim

Semantics is easy.

Command: order to do something

Study the book until you have understood it.

Expressions are *evaluated*, commands are *executed*

Commands

Concrete syntax:

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

Commands

Abstract syntax:

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

Com.thy

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

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

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

Examples: derivation trees

$$\frac{\vdots}{(''x'' ::= N\ 5;\ ''y'' ::= V\ ''x'',\ s) \Rightarrow\ ?} \qquad \frac{\vdots}{(w,\ s_i) \Rightarrow\ ?}$$

where
$$\begin{aligned} w &= WHILE\ b\ DO\ c \\ b &= NotEq\ (V\ ''x'')\ (N\ 2) \\ c &= ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1) \\ s_i &= s(''x'' := i) \end{aligned}$$

$NotEq\ a_1\ a_2 =$
$Not(And\ (Not(Less\ a_1\ a_2))\ (Not(Less\ a_2\ a_1)))$

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 $w = WHILE\ b\ DO\ c$?

Automating rule inversion

Isabelle command **inductive_cases** produces theorems that perform rule inversions automatically.

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 $\wedge$!

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 $asm_i \Longrightarrow P$)

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*, *fastforce* 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 $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$
$iw = \mathit{IF}\ b\ \mathit{THEN}\ c;\ w\ \mathit{ELSE}\ \mathit{SKIP}$

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) does not terminate iff $\neg\ (\exists\, t.\ (c,\ s) \Rightarrow\ t)$?

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:

$$(com,state) \rightarrow (com,state)$$

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

> *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:

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

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 $\neg\ (\exists\, cs'.\ cs \to cs')$

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

$$(''z'' ::= V\ ''x''; ''x'' ::= V\ ''y''; ''y'' ::= V\ ''z'', s) \to \ldots$$

where $s = <''x'' := 3, ''y'' := 7, ''z'' := 5>$.

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

where
$$\begin{array}{lcl} w & = & WHILE\ b\ DO\ c \\ b & = & Less\ (V\ ''x'')\ (N\ 1) \\ c & = & ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1) \\ s_n & = & <''x'' := n> \end{array}$$

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:

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

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

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

(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

$\rightarrow$ is deterministic:

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

Therefore: no difference between

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

must terminate (all $\rightarrow$ paths terminate)

Therefore: $\Rightarrow$ correctly reflects termination behaviour.

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

Stack Machine

Instructions:

datatype $instr =$

$LOADI\ int$	load value
$\mid LOAD\ vname$	load var
$\mid ADD$	add top of stack
$\mid STORE\ vname$	store var
$\mid JMP\ int$	jump
$\mid JMPLESS\ int$	jump if $<$
$\mid JMPGE\ int$	jump if $\geq$

Semantics

Type synonyms:

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

Execution of 1 instruction:

$$iexec :: instr \Rightarrow config \Rightarrow config$$

Instruction execution

$iexec\ instr\ (i,\ s,\ stk) =$
$(\textsf{case}\ instr\ \textsf{of}\ LOADI\ n \Rightarrow (i + 1,\ s,\ n \# stk)$
$\mid LOAD\ x \Rightarrow (i + 1,\ s,\ s\ x \# stk)$
$\mid ADD \Rightarrow (i + 1,\ s,\ (hd2\ stk + hd\ stk) \# tl2\ stk)$
$\mid STORE\ x \Rightarrow (i + 1,\ s(x := hd\ stk),\ tl\ stk)$
$\mid JMP\ n \Rightarrow (i + 1 + n,\ s,\ stk)$
$\mid JMPLESS\ n \Rightarrow$
$(\textsf{if}\ hd2\ stk < hd\ stk\ \textsf{then}\ i + 1 + n\ \textsf{else}\ i + 1,$
$s,\ tl2\ stk)$
$\mid JMPGE\ n \Rightarrow$
$(\textsf{if}\ hd\ stk \le hd2\ stk\ \textsf{then}\ i + 1 + n\ \textsf{else}\ i + 1,$
$s,\ tl2\ stk))$

Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$instr\ list \vdash config \to config$$

$$
\begin{aligned}
&P \vdash c \to c' = \\
&(\exists\, i\ s\ stk. \\
&\quad c = (i,\ s,\ stk) \wedge \\
&\quad c' = iexec\ (P\ !!\ i)\ (i,\ s,\ stk) \wedge \\
&\quad 0 \le i \wedge i < isize\ P)
\end{aligned}
$$

where $'a\ list\ !!\ int$ = nth instruction of list
and $isize :: list \Rightarrow int$ = list size as integer

Program execution ($*$ steps)

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:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

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

Proof by induction on a (with arbitrary stk).

Needs lemmas!

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

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

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

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

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

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)

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

Example

Let $b = And \; (Less \; (V \; ''x'') \; (V \; ''y''))$
$\quad (Not \; (Less \; (V \; ''z'') \; (V \; ''a'')))$.

$bcomp \; b \; False \; 3 =$

$[LOAD \; ''x''$,
$LOAD \; ''y''$,

,
$LOAD \; ''z''$,
$LOAD \; ''a''$,

]

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

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

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

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

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

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

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

Correctness of $bcomp$

$$0 \leq n \Longrightarrow$$
$$bcomp\ b\ c\ n$$
$$\vdash (0,\ s,\ stk) \rightarrow*$$
$$(isize\ (bcomp\ b\ c\ n) + (\textit{if}\ c = bval\ b\ s\ \textit{then}\ n\ \textit{else}\ 0),$$
$$s,\ stk)$$

Compiling *com*

$$ccomp :: com \Rightarrow instr\ list$$

$$ccomp\ SKIP = []$$

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

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

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

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

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

$$\textsf{let}\ cc = ccomp\ c;$$
$$\quad cb = bcomp\ b\ False\ (isize\ cc + 1)$$
$$\textsf{in}\ cb\ @\ cc\ @\ [JMP\ (-\ (isize\ cb + isize\ cc + 1))]$$

Correctness of $ccomp$

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

$$(c,\ s) \Rightarrow t \Longrightarrow$$
$$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow* (isize\ (ccomp\ c),\ t,\ stk)$$

Proof by rule induction.

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

If $ccomp\ c$ with start state s produces result t,
and if(!) $(c,\ s) \Rightarrow t'$, then "$\Longrightarrow$" implies
that $ccomp\ c$ with start state s 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.

The other direction

Two approaches:

- In the absence of nondeterminism:
  Prove that $ccomp$ preserves nontermination.
  A nice proof of this fact requires *coinduction*.
  Isabelle supports coinduction, this course avoids it.
- A direct proof: `Comp_Rev.thy`

$$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow\!\!*\ (isize\ (ccomp\ c),\ t,\ stk') \Longrightarrow (c,\ s) \Rightarrow t$$

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: "TypeError: ..." in Python.

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:

$state = vname \Rightarrow val$

Arithmetic expresssions:

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

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 recursive 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 $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

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

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

The functional alternative

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

Exercise!

Boolean expressions

Syntax as before. Semantics:

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

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

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

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

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

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

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

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

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

The other rules remain unchanged.

Example

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

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

10 A Typed Version of IMP

Type system

There are two types:

datatype $ty = Ity \mid Rty$

What is the type of $Plus\ (V\ ''x'')\ (V\ ''y'')$?

Depends on the type of $V\ ''x''$ and $V\ ''y''$!

A *type environment* maps variable names to their types:

$tyenv = vname \Rightarrow ty$

The type of an expression is always relative to a type enviroment Γ. Standard notation:

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

Read: *In the context of* Γ, e *has type* τ

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

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

where $\Gamma\ ''x'' = \mathit{Ity}$.

Well-typed $bexp$

Notation:

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

Read: *In context Γ, b is well-typed*.

The rules:

$$\Gamma \vdash Bc\ v$$

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

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

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

Example: $\Gamma \vdash Less\ (Ic\ i)\ (Rc\ r)$ does not hold.

Well-typed commands

Notation:

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

Read: *In context* Γ*,* c *is well-typed*.

The rules:

$$\Gamma \vdash \mathit{SKIP} \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

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

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

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

Syntax-directedness

All three sets of typing rules are *syntax-directed*:

> *There is exactly one rule for each syntactic construct (eg $SKIP$, $::=$ etc).*

Therefore each set of rules is executable without backtracking:

> *Given Γ and a term $a/b/c$, its well-typedness (and its type) is computable by backchaining without backtracking.*

The big and small step semantics are not syntax-directed.

Compositionality

All three sets of typing rules are *compositional*:

> *Well-typedness of a syntactic construct $C\ t_1 \ldots t_n$ depends only on the well-typedness of* $t_1, \ldots, t_n$.

Therefore type-checking always terminates and requires at most as many backchaining steps as the size of the term.

The big step semantics is not compositional because the execution of $WHILE$ depends on the execution of $WHILE$.

Well-typed states

Even well-typed programs can get stuck . . .
. . . if they start in an unsuitable state.

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

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

The type of a value:

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

Well-typed state:

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

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:

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

Preservation:

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

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

Type soundness:

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

Progress:

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

$aexp$

Progress:

$$\llbracket \Gamma \vdash a : \tau;\ \Gamma \vdash s \rrbracket \Longrightarrow \exists\, v.\ taval\ a\ s\ v$$

Preservation:

$$\llbracket \Gamma \vdash a : \tau;\ taval\ a\ s\ v;\ \Gamma \vdash s \rrbracket \Longrightarrow type\ v = \tau$$

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 III

Data-Flow Analyses and Optimization

11 Definite Initialization Analysis

12 Live Variable Analysis

13 Information Flow Analysis

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.

Examples: ok or not?

Assume $''x''$ is initialized:

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

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

Assume $''x''$ and $''y''$ are initialized:

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

Simplifying principle

We do not analyze boolean expressions to determine program execution.

11 Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics

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

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

$vars :: bexp \Rightarrow vname\ set$
$vars\ (Bc\ v) = \{\}$
$vars\ (Not\ b) = vars\ b$
$vars\ (And\ b_1\ b_2) = vars\ b_1 \cup vars\ b_2$
$vars\ (Less\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

Vars.thy

11 Definite Initialization Analysis

Modified example from the JLS:

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

Similar statements for each each language construct.

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

$D\ A\ c\ A'$ 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.

$$D\ A\ SKIP\ A$$

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

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

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

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

Correctness of D

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

  $\Longrightarrow$ We need a new, finer-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 Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics

$$state = vname \Rightarrow val\ option$$

where

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

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

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

Expression evaluation

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

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

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

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

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

$bval\ (Bc\ v)\ s = Some\ v$

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

$bval\ (And\ b_1\ b_2)\ s =$
$(\mathsf{case}\ (bval\ b_1\ s,\ bval\ b_2\ s)\ \mathsf{of}$
$\quad (Some\ bv_1,\ Some\ bv_2) \Rightarrow Some(bv_1 \wedge bv_2)$
$\mid \_ \Rightarrow None)$

$bval\ (Less\ a_1\ a_2)\ s =$
$(\mathsf{case}\ (aval\ a_1\ s,\ aval\ a_2\ s)\ \mathsf{of}$
$\quad (Some\ i_1,\ Some\ i_2) \Rightarrow Some(i_1 < i_2)$
$\mid \_ \Rightarrow None)$

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.

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

We need to prove a generalized statement:

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

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

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.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is x initially dead or live? $(x \neq y)$

$x ::= N\ 0$ ☹

$y ::= V\ x;\ y ::= N\ 0;\ x ::= N\ 0$ ☺

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

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

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

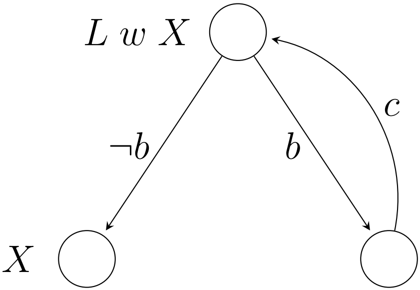
$$L\ c\ X \ = \ \text{live before } c \text{ relative to } X$$

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & X - \{x\} \cup vars\ a \\ L\ (c_1;\ c_2)\ X & = & (L\ c_1 \circ L\ c_2)\ X \\ \multicolumn{3}{l}{L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X \ =} \\ \multicolumn{3}{l}{\quad vars\ b \cup L\ c_1\ X \cup L\ c_2\ X} \end{array}$$

Example:

$$L\ (''y'' ::= V\ ''z'';\ ''x'' ::= Plus\ (V\ ''y'')\ (V\ ''z''))$$
$$\{''x''\} \ = \ \{''z''\}$$

$WHILE\ b\ DO\ c$



$L\ w\ X$ must satisfy

$$\begin{array}{lcll} vars\ b & \subseteq & L\ w\ X & \text{(evaluation of } b\text{)} \\ X & \subseteq & L\ w\ X & \text{(exit)} \\ L\ c\ (L\ w\ X) & \subseteq & L\ w\ X & \text{(execution of } c\text{)} \end{array}$$

We define

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

$\Longrightarrow$

$vars\ b \subseteq L\ w\ X$ ✓

$X \subseteq L\ w\ X$ ✓

$L\ c\ (L\ w\ X) \subseteq L\ w\ X$?

$$
\begin{array}{lcl}
L\ SKIP\ X & = & X \\
L\ (x ::= a)\ X & = & X - \{x\} \cup vars\ a \\
L\ (c_1;\ c_2)\ X & = & (L\ c_1 \circ L\ c_2)\ X
\end{array}
$$

$$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =\ vars\ b \cup L\ c_1\ X \cup L\ c_2\ X$$

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

Example:

$$L\ (WHILE\ Less\ (V\ ''x'')\ (V\ ''x'')\ DO\ ''y'' ::= V\ ''z'')$$
$$\{''x''\}\ =\ \{''x'', ''z''\}$$

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.

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

Many standard data-flow analyses are gen/kill.
In particular liveness analysis.

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

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

$gen :: com \Rightarrow vname\ set$

$$
\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & vars\ a \\
gen\ (c_1;\ c_2) & = & gen\ c_1 \cup (gen\ c_2 - kill\ c_1)
\end{array}
$$

$$gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$$
$$\quad vars\ b \cup gen\ c_1 \cup gen\ c_2$$

$$gen\ (WHILE\ b\ DO\ c) = vars\ b \cup gen\ c$$

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

Proof by induction on c.

$\Longrightarrow$

$$L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

Digression: definite initialization via gen/kill

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

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

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

$$kill\ c\ =\ \{\}$$

12 Live Variable Analysis

Soundness of L

Dead Variable Elimination

True Liveness

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.
For the two $WHILE$ cases we do not need the definition of $L\ w$ but only the characteristic property

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

Optimality of $L\ w$

The result of L should be as small as possible: the more dead variables, the better (for program optimization).

> $L\ w\ X$ *should be the* *least* *set such that*
> $vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$.

Follows easily from $L\ c\ X = X - kill\ c \cup gen\ c$:

> $vars\ b \cup X \cup L\ c\ P \subseteq P \Longrightarrow$
> $L\ (WHILE\ b\ DO\ c)\ X \subseteq P$

12 Live Variable Analysis

Soundness of L

Dead Variable Elimination

True Liveness

Comparisons

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$
\begin{aligned}
bury\ SKIP\ X &= SKIP \\
bury\ (x ::= a)\ X &= \textsf{\textit{if}}\ x \in X\ \textsf{\textit{then}}\ x ::= a\ \textsf{\textit{else}}\ SKIP \\
bury\ (c_1;\ c_2)\ X &= bury\ c_1\ (L\ c_2\ X);\ bury\ c_2\ X \\
bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X &= \\
&\quad IF\ b\ THEN\ bury\ c_1\ X\ ELSE\ bury\ c_2\ X \\
bury\ (WHILE\ b\ DO\ c)\ X &= \\
&\quad WHILE\ b\ DO\ bury\ c\ (vars\ b \cup X \cup L\ c\ X)
\end{aligned}
$$

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.

$$(bc_1;\ bc_2 = bury\ c\ X) =$$
$$(\exists\, c_1\ c_2.$$
$$c = c_1;\ c_2\ \wedge$$
$$bc_2 = bury\ c_2\ X \wedge bc_1 = bury\ c_1\ (L\ c_2\ X))$$

Terminology

Let $f :: t \Rightarrow t$ and $x :: t$.

If $f\ x = x$ then x is a *fixed point* of f.

Let $\leq$ be a partial order on t, eg $\subseteq$ on sets.

If $f\ x \leq x$ then x is a *post-fixed point* (*pfp*) of f.

Application to $L\ w$

Remember the specification of $L\ w$:

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

This is the same as saying that $L\ w\ X$ should be a pfp of

$$\lambda P.\ vars\ b \cup X \cup L\ c\ P$$

and in particular of $L\ c$.

True liveness

$L\ (''x'' ::= V\ ''y'')\ \{\} = \{''y''\}$

But $''y''$ is not truly live: it is assigned to a dead variable.

Problem: $L\ (x ::= a)\ X = X - \{x\} \cup vars\ a$

Better:

$L\ (x ::= e)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ X - \{x\} \cup vars\ e\ \textit{else}\ X)$

But then

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

is not correct anymore.

$L\ (x ::= e)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ X - \{x\} \cup \mathit{vars}\ e\ \textit{else}\ X)$

$L\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X = \mathit{vars}\ b \cup X \cup L\ c\ X$

Let $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$
where $b\ =\ \mathit{Less}\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;\ x ::= V\ z$
and $\mathit{distinct}\ [x,\ y,\ z]$

Then $L\ w\ \{y\} = \{x,\ y\}$, but z is live before w !

$\{x\}\ \ y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$

$\Longrightarrow\ L\ w\ \{y\} = \{y\} \cup \{y\} \cup \{x\}$

$$b = \mathit{Less}\ (N\ 0)\ (V\ y)$$
$$c = y ::= V\ x;\ x ::= V\ z$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$$\{x,\ z\}\ \ y ::= V\ x\ \ \{y,\ z\}\ x ::= V\ z\ \ \{x,\ y\}$$

$$L\ c\ \{x,\ y\} = \{x,\ z\} \not\subseteq \{x,\ y\}$$

$L\ w$ for true liveness

Define $L\ w\ X$ *as the least pfp of*
$\lambda P.\ vars\ b \cup X \cup L\ c\ P$

Existence of least fixed points

Theorem (Knaster-Tarski) Let $f :: t\ set \Rightarrow t\ set$.
If f is monotone $(X \subseteq Y \Longrightarrow f(X) \subseteq f(Y))$
then

$$lfp(f) := \bigcap\{P \mid f(P) \subseteq P\}$$

is the least fixed and post-fixed point of f.

Proof of Knaster-Tarski

$$lfp(f) := \bigcap\{P \mid f(P) \subseteq P\}$$

- $f\ (lfp\ f) \subseteq lfp\ f$
- $lfp\ f$ is the least post-fixed point of f
- $lfp\ f \subseteq f\ (lfp\ f)$
- $lfp\ f$ is the least fixed point of f

Definition of L

$L\ (x ::= e)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ X - \{x\} \cup vars\ e\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = lfp\ f_w$
where $f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

Lemma $L\ c$ is monotone.

Proof by induction on c using that lfp is monotone:
$lfp\ f \subseteq lfp\ g$ if for all X, $f\ X \subseteq g\ X$

Corollary f_w is monotone.

Computation of lfp

Theorem Let $f :: t\ set \Rightarrow t\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and the chain $\{\} \subseteq f(\{\}) \subseteq f(f(\{\})) \subseteq \dots$ stabilizes after a finite number of steps, i.e. $f^{k+1}(\{\}) = f^k(\{\})$ for some k,

then $\mathit{lfp}(f) = f^k(\{\})$.

Proof Show $f^i(\{\}) \subseteq p$ for any pfp p of f (by induction on i).

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables read in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Let $V_w = \mathit{vars}\ b \cup \mathit{vars}\ c \cup X$

Corollary $P \subseteq V_w \Longrightarrow f_w\ P \subseteq V_w$

Hence $f_w^k\ \{\}$ stabilizes for some $k \leq |V_w|$.
More precisely: $k \leq |\mathit{vars}\ c| + 1$
because $f_w\ \{\} \supseteq \mathit{vars}\ b \cup X$.

Example

Let $w = \textit{WHILE } b \textit{ DO } c$
where $b \;=\; \textit{Less } (N\ \mathit{0})\ (V\ y)$
and $c \;=\; y ::= V\ x;\ x ::= V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

$$\{\}\ \ y ::= V\ x\ \ \{\}\ \ x ::= V\ z\ \ \{\}$$

$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x,\ y\}$:

$$\{x\}\ \ y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$$

$f_w\ \{x,\ y\} = \{y\} \cup L\ c\ \{x{,}y\} = \{x,\ y,\ z\}$:

$$\{x,\ z\}\ \ y ::= V\ x\ \ \{y,\ z\}\ \ x ::= V\ z\ \ \{x,\ y\}$$

Computation of lfp in Isabelle

From the library theory `While_Combinator`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = (\textsf{if}\ b\ s\ \textsf{then}\ while\ b\ f\ (f\ s)\ \textsf{else}\ s)$

Lemma Let $f :: t\ set \Rightarrow t\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and bounded by some finite set C:
  $X \subseteq C \Longrightarrow f\ X \subseteq C$

then $lfp\ f = while\ (\lambda X.\ f\ X \neq X)\ f\ \{\}$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH) $\Longrightarrow$
$\forall P.\ f_w\ P \subseteq g_w\ P \Longrightarrow f_w(g_w^i\ \{\}) = g_w\ (g_w^i\ \{\}) = g_w^i\ \{\}$
$\Longrightarrow L\ w\ X = lfp\ f_w \subseteq g_w^i\ \{\} = Lb\ w\ X$

If $Lb\ w\ X = V_w$: $L\ w\ X \subseteq V_w$ (by Lemma)

12 Live Variable Analysis

Soundness of L

Dead Variable Elimination

True Liveness

Comparisons

Comparison of analyses

- Definite initialization 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.

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 Analysis

Security Levels

Security levels:

type_synonym $level = nat$

Every variable has a security level:

$sec :: vname \Rightarrow level$

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

$sec\ x = length\ x$

Security Levels on $aexp$

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

$sec :: aexp \Rightarrow level$

$sec\ (N\ n) = 0$
$sec\ (V\ x) = sec\ x$
$sec\ (Plus\ a\ b) = max\ (sec\ a)\ (sec\ b)$

Security Levels on $bexp$

$sec :: bexp \Rightarrow level$

$$sec\ (Bc\ v) = 0$$
$$sec\ (Not\ b) = sec\ b$$
$$sec\ (And\ b_1\ b_2) = max\ (sec\ b_1)\ (sec\ b_2)$$
$$sec\ (Less\ a\ b) = max\ (sec\ a)\ (sec\ b)$$

Security Levels on States

Agreement of states up to a certain level:

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

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

Noninterference lemmas for expressions:

$$\frac{s_1 = s_2 \; (\leq l) \qquad sec\ a \leq l}{aval\ a\ s_1 = aval\ a\ s_2}$$

$$\frac{s_1 = s_2 \; (\leq l) \qquad sec\ b \leq l}{bval\ b\ s_1 = bval\ b\ s_2}$$

Security Type System

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?

Let $c = \;\; \mathit{IF\ Less}\ (V\ ''x1'')\ (V\ ''x'')$
$\mathit{THEN}\ ''x1'' ::= N\ 0$
$\mathit{ELSE}\ ''x1'' ::= N\ 1$

$1 \vdash c$? Yes

$2 \vdash c$? Yes

$3 \vdash c$? No

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

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

Proof by ... as usual.

13 Information Flow Analysis

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:

$$l \vdash' \mathit{SKIP}$$

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

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

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

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

- The subsumption-based system $\vdash'$ is neither syntax-directed nor directly executable.
- Need to guess when to use the subsumption rule.

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 $\vdash$.

13 Information Flow Analysis

- Secure IMP
- A Security Type System
- A Type System with Subsumption
- A Bottom-Up Type System
- Beyond

- 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: $\mathit{0} \vdash' {''x''} ::= N\ \mathit{1}$ but not $\vdash {''x''} ::= N\ \mathit{1} : \mathit{0}$

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

Proof by induction.

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!

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

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 IV

Hoare Logic

14 Partial Correctness

15 Verification Conditions

16 Total Correctness

14 Partial 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:

$''x'' ::= N\ 0;\ ''y'' ::= N\ 0;\ w\ n$

where

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

At the end of the execution,
variable $''x''$ should contain the sum $1 + \ldots + n$.

A proof via operational semantics

Theorem:

$$(''x'' ::= N\ 0;\ ''y'' ::= N\ 0;\ w\ n,\ s) \Rightarrow t \Longrightarrow$$
$$t\ ''x'' = \sum \{1..n\}$$

Required Lemma:

$$(w\ n,\ s) \Rightarrow t \Longrightarrow$$
$$t\ ''x'' = s\ ''x'' + \sum \{s\ ''y'' + 1..n\}$$

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.

14 Partial Correctness

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

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

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

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

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

$$\vdash \{P\}\ x ::= Plus\ (V\ x)\ (N\ 5)\ \{\lambda t.\ t\ x = 5\}$$

where $P = (\lambda s.\ (\lambda t.\ t\ x = 5)(s[Plus\ (V\ x)\ (N\ 5)/x]))$
$= (\lambda s.\ (\lambda t.\ t\ x = 5)(s(x := s\ x + 5)))$
$= (\lambda s.\ s\ x + 5 = 5)$

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

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

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

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

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

$$wp\ SKIP\ Q = Q$$

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

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

$$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q =$$
$$(\lambda s.\ (bval\ b\ s \longrightarrow wp\ c_1\ Q\ s) \wedge$$
$$(\neg\ bval\ b\ s \longrightarrow wp\ c_2\ Q\ s))$$

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

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

Completeness

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

Proof idea: do not prove $\vdash \{P\}\ c\ \{Q\}$ directly, prove something stronger:

Lemma $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$
Proof by induction on c, for arbitary Q.

Now prove $\vdash \{P\}\ c\ \{Q\}$ from $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$ by the consequence rule because

Fact $\models \{P\}\ c\ \{Q\} \Longrightarrow \forall s.\ P\ s \longrightarrow wp\ c\ Q\ s$
Follows directly from defs of $\models$ and wp.

$$\vdash \{P\}\ c\ \{Q\} \quad \longleftrightarrow \quad \models \{P\}\ c\ \{Q\}$$

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

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

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 an important research problem.

Which we ignore for the moment.

But come back to later.

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 preconditions: $pre :: com \Rightarrow assn \Rightarrow assn$
   - 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, except for While:

$$\textbf{datatype}\ \mathit{acom} = \mathit{ASKIP} \mid \mathit{Aassign}\ \mathit{vname}\ \mathit{aexp} \mid \mathit{Aseq}\ \mathit{acom}\ \mathit{acom} \mid \mathit{Aif}\ \mathit{bexp}\ \mathit{acom}\ \mathit{acom} \mid \mathit{Awhile}\ \mathit{assn}\ \mathit{bexp}\ \mathit{acom}$$

Concrete syntax: like commands, except for WHILE:

$$\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c$$

Weakest precondition

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

$$pre\ ASKIP\ Q = Q$$

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

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

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

$$pre\ (\{I\}\ WHILE\ b\ DO\ c)\ Q = I$$

Warning

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

Verification condition

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

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

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

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

$vc\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q =$
$(\lambda s.\ vc\ c_1\ Q\ s \wedge vc\ c_2\ Q\ s)$

$vc\ (\{I\}\ WHILE\ b\ DO\ c)\ Q =$
$(\lambda s.\ (I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s) \wedge$
$(I\ s \wedge bval\ b\ s \longrightarrow pre\ c\ I\ s) \wedge vc\ c\ I\ s)$

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

$$strip :: acom \Rightarrow com$$

$$strip\ ASKIP = SKIP$$
$$strip\ (x ::= a) = x ::= a$$
$$strip\ (c_1;\ c_2) = strip\ c_1;\ strip\ c_2$$
$$strip\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$$
$$IF\ b\ THEN\ strip\ c_1\ ELSE\ strip\ c_2$$
$$strip\ (\{I\}\ WHILE\ b\ DO\ c) = WHILE\ b\ DO\ strip\ c$$

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

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

Proof by induction on c, for arbitrary Q.

Corollary:

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

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

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

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

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

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. $\vdash$

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

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 V

Abstract Interpretation

- Abstract interpretation
  is a generic approach to static program analysis.
- It subsumes and improves our earlier approaches.
- Aim: For each program point, compute the possible values of all variables
- Method: Execute/interpret program with abstract instead of concrete values, eg intervals instead of numbers.

Applications: Optimization

- Constant folding
- Unreachable and dead code elimination
- Array access optimization:
  `a[i] := 1; a[j] := 2; x := a[i]` $\rightsquigarrow$
  `a[i] := 1; a[j] := 2; x := 1`
  if $i \neq j$
- ...

Applications: Debugging/Verification

Detect presence or absence of certain runtime exceptions/errors:

- Interval analysis: $i \in [m, n]$:
  - No division by 0 in `e/i` if $0 \notin [m, n]$
  - No ArrayIndexOutOfBoundsException in `a[i]` if $0 \leq m \land n <$ `a.length`
  - ...
- Null pointer analysis
- ...

Precision

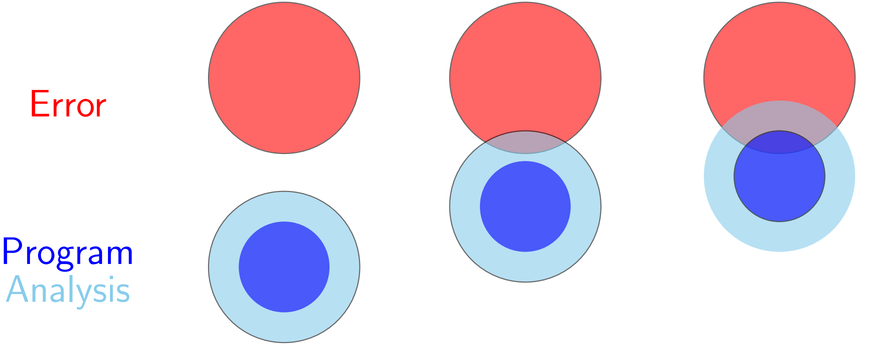
A consequence of Rice's theorem:

> *In general, the possible values of a variable cannot be computed precisely.*

Program analyses overapproximate: they compute a *superset* of the possible values of a variable.

If an analysis says that some value/error/exception

- cannot arise, this is definitely the case.
- can arise, this is only potentially the case.
  Beware of *false alarms* because of overapproximation.


Error
Program
Analysis
No Alarm
False Alarm
True Alarm

Annotated commands

Like in Hoare logic, we annotate

$$\{\ \dots\ \}$$

program text with semantic information.

Not just loops but also all intermediate program points, for example:

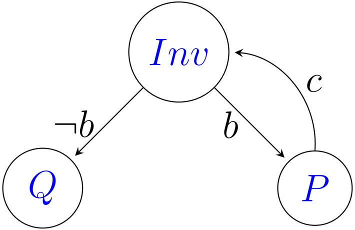
$$\texttt{x := 0}\ \{\ \dots\ \};\ \texttt{y := 0}\ \{\ \dots\ \}$$

Annotated `WHILE`

View

$$\{Inv\}$$
$$\texttt{WHILE}\ b\ \texttt{DO}\ \ \{\mathsf{P}\}\ \ c$$
$$\{Q\}$$

as a *control flow graph*



with annotated nodes

The starting point: Collecting Semantics

Collects all possible states for each program point:

`x := 0` $\{ <x := 0> \}$ `;`
$\{ <x := 0>, <x := 2>, <x := 4> \}$
`WHILE x < 3`
`DO` $\{ <x := 0>, <x := 2> \}$
  `x := x+2` $\{ <x := 2>, <x := 4> \}$
$\{ <x := 4> \}$

Infinite sets of states:

$\{\ldots, <x := -1>, <x := 0>, <x := 1>, \ldots\}$
`WHILE x < 3`
`DO` $\{\ \ldots, <x := 1>, <x := 2>\ \}$
`  x := x+2` $\{\ \ldots, <x := 3>, <x := 4>\ \}$
$\{\ <x := 3>, <x := 4>, \ldots\ \}$

Multiple variables:

`x := 0; y := 0` $\{\ <x{:=}0,\ y{:=}0>\ \}$ `;`

$\{\ <x{:=}0,\ y{:=}0>,\ <x{:=}2,\ y{:=}1>,\ <x{:=}4,\ y{:=}2>\ \}$

`WHILE x < 3`

`DO` $\{\ <x{:=}0,\ y{:=}0>,\ <x{:=}2,\ y{:=}1>\ \}$

  `x := x+2; y := y+1`

  $\{\ <x{:=}2,\ y{:=}1>,\ <x{:=}4,\ y{:=}2>\ \}$

$\{\ <x{:=}4,\ y{:=}2>\ \}$

A first approximation

$$(vname \Rightarrow val)\ set \quad \rightsquigarrow \quad vname \Rightarrow val\ set$$

```
x := 0 { <x := {0}> } ;
{ <x := {0,2,4}> }
WHILE x < 3
DO { <x := {0,2}> }
   x := x+2 { <x := {2,4}> }
{ <x := {4}> }
```

Loses relationships between variables
but simplifies matters a lot.

Example:

$\{ <x{:=}0,\ y{:=}0>,\ <x{:=}1, y{:=}1>\ \}$

is approximated by

$<x{:=}\{0,1\},\ y{:=}\{0,1\}>$

which also subsumes

$<x{:=}0,\ y{:=}1>$ and $<x{:=}1, y{:=}0>$.

Abstract Interpretation

Approximate sets of concrete values by *abstract values*

Example: approximate sets of numbers by intervals

Execute/interpret program with abstract values

Example

Consistently annotated program:

```
x := 0 { <x := [0,0]> } ;
{ <x := [0,4]> }
WHILE x < 3
DO { <x := [0,2]> }
   x := x+2 { <x := [2,4]> }
{ <x := [3,4]> }
```

The annotations are computed by

- starting from an un-annotated program and
- iterating abstract execution
- until the annotations stabilize.

```
x := 0

WHILE x < 3
DO
  x := x+2
```

Concrete syntax

$$
\begin{array}{lcl}
'a\ acom & ::= & SKIP\ \{\ 'a\ \}\ \mid\ string ::= aexp\ \{\ 'a\ \} \\
& \mid & 'a\ acom\ ;\ 'a\ acom \\
& \mid & IF\ bexp\quad THEN\ \{\ 'a\ \}\ \ 'a\ acom \\
& & \qquad\qquad\quad ELSE\ \{\ 'a\ \}\ \ 'a\ acom \\
& & \{\ 'a\ \} \\
& \mid & \{\ 'a\ \} \\
& & WHILE\ bexp\ DO\ \ \{\ 'a\ \}\ \ 'a\ acom \\
& & \{\ 'a\ \}
\end{array}
$$

$'a$: type of annotations

Example: $''x'' ::= N\ 1\ \{9\};\ SKIP\ \{6\}\ \ ::\ \ nat\ acom$

Abstract syntax

datatype

$$
\begin{aligned}
&'a\ acom = \\
&\quad\ \ SKIP\ \ 'a \\
&\quad |\ Assign\ \ string\ \ aexp\ \ 'a \\
&\quad |\ Seq\ \ ('a\ acom)\ \ ('a\ acom) \\
&\quad |\ If\ \ bexp\ \ 'a\ \ ('a\ acom)\ \ 'a\ \ ('a\ acom)\ \ 'a \\
&\quad |\ While\ \ 'a\ \ bexp\ \ 'a\ \ ('a\ acom)\ \ 'a
\end{aligned}
$$

Auxiliary functions: $post$

$post :: {'a}\ acom \Rightarrow {'a}$

$post\ (SKIP\ \{P\}) = P$

$post\ (x ::= e\ \{P\}) = P$

$post\ (C_1;\ C_2) = post\ C_2$

$post\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) = Q$

$post\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = Q$

Auxiliary functions: $strip$

$strip :: {}'a\ acom \Rightarrow com$

$strip\ (SKIP\ \{P\}) = SKIP$

$strip\ (x ::= e\ \{P\}) = x ::= e$

$strip\ (C_1;\ C_2) = strip\ C_1;\ strip\ C_2$

$strip\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{P\})$
$\quad = IF\ b\ THEN\ strip\ C_1\ ELSE\ strip\ C_2$

$strip\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\})$
$\quad = WHILE\ b\ DO\ strip\ C$

We call C and C' *strip-equal* iff $strip\ C = strip\ C'$.

Auxiliary functions: $anno$

$anno :: {}'a \Rightarrow com \Rightarrow {}'a\ acom$

$anno\ A\ SKIP = SKIP\ \{A\}$

$anno\ A\ (x ::= e) = x ::= e\ \{A\}$

$anno\ A\ (c_1;\ c_2) = anno\ A\ c_1;\ anno\ A\ c_2$

$anno\ A\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$
$\quad IF\ b\ THEN\ \{A\}\ anno\ A\ c_1\ ELSE\ \{A\}\ anno\ A\ c_2$
$\{A\}$

$anno\ A\ (WHILE\ b\ DO\ c) =$
$\quad \{A\}\ WHILE\ b\ DO\ \{A\}\ anno\ A\ c\ \{A\}$

Auxiliary functions: map_acom

$map\_acom :: ('a \Rightarrow 'b) \Rightarrow 'a\ acom \Rightarrow 'b\ acom$

$map\_acom\ f\ C$ applies f to all annotations in C

Auxiliary functions: $annos$

$annos :: 'a\ acom \Rightarrow 'a\ list$

$annos\ C$ is the list (in some order) of annotations of C

Annotate commands with the set of states that can occur at each annotation point.

The annotations are generated iteratively:

$step :: state\ set \Rightarrow state\ set\ acom \Rightarrow state\ set\ acom$

Each step executes all atomic commands simultaneously, propagating the annotations one step further.

start states

flowing into the command

$step$

$$step\ S\ (SKIP\ \{\_\}) = SKIP\ \{S\}$$

$$step\ S\ (x ::= e\ \{\_\}) =$$
$$x ::= e\ \{\{s(x := aval\ e\ s)\ |s.\ s \in S\}\}$$

$$step\ S\ (C_1;\ C_2) = \ step\ S\ C_1;\ step\ (post\ C_1)\ C_2$$

$$step\ S\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{\_\}) =$$
$$IF\ b\ THEN\ \{\{s \in S.\ bval\ b\ s\}\}\ step\ P_1\ C_1$$
$$ELSE\ \{\{s \in S.\ \neg\ bval\ b\ s\}\}\ step\ P_2\ C_2$$
$$\{post\ C_1 \cup post\ C_2\}$$

$step$

$$
\begin{array}{l}
step\ S\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{\_\}) = \\
\{S \cup post\ C\} \\
WHILE\ b \\
DO\ \{\{s \in I.\ bval\ b\ s\}\} \\
\quad\quad step\ P\ C \\
\{\{s \in I.\ \neg\ bval\ b\ s\}\}
\end{array}
$$

Collecting semantics

View command as a control flow graph

- where you constantly feed in some fixed input set S (typically all possible states)
- and pump/propagate it around the graph
- until the annotations stabilize —
  this may happen in the limit only!

Stabilization means fixed point:

$$step\ S\ C = C$$

Collecting_Examples.thy

Abstract example

Let $C = \ \{\ I\ \}$
$\texttt{WHILE}\ b$
$\texttt{DO}\ \{\ P\ \}\ C_0$
$\{\ Q\ \}$

$step\ S\ C = C$ means

$$\begin{array}{lcl} I & = & S \cup post\ C_0 \\ P & = & \{s \in I.\ bval\ b\ s\} \\ C_0 & = & step\ P\ C_0 \\ Q & = & \{s \in I.\ \neg\ bval\ b\ s\} \end{array}$$

Fixed point = solution of equation system

Iteration is just one way of solving equations

Why *least* fixed point?

$\{ I \}$
WHILE true
DO $\{ I \}$ SKIP $\{ I \}$
$\{ \{\} \}$

Is fixed point of $step\ \{\}$ for every I

But the "reachable" fixed point is $I = \{\}$

Complete lattice

Definition
A type $'a$ with a partial order $\leq$ is a *complete lattice* if every set $S :: 'a\ set$ has a *greatest lower bound* $l :: 'a$:

- $\forall\, s \in S.\ l \leq s$
- If $\forall\, s \in S.\ l' \leq s$ then $l' \leq l$

The greatest lower bound (*infimum*) of S is often denoted by $\bigsqcap S$.

Fact Type $'a\ set$ is a complete lattice where $\bigcap$ is the infimum.

Lemma In a complete lattice, every set S of elements also has a *least upper bound* (*supremum*) $\bigsqcup S$:

- $\forall\, s \in S.\ s \leq \bigsqcup S$
- If $\forall\, s{\in}S.\ s \leq u$ then $\bigsqcup S \leq u$

The least upper bound is the greatest lower bound of all upper bounds: $\bigsqcup S = \bigsqcap \{u.\ \forall\, s \in S.\ s \leq u\}$.

Thus complete lattices can be defined via the existence of all infima or all suprema or both.

Existence of least fixed points

Definition A function f on a partial order $\leq$ is *monotone* if $x \leq y \Longrightarrow f\,x \leq f\,y$.

Theorem (Knaster-Tarski) Every monotone function on a complete lattice has the least (post-)fixed point

$$\prod \{p.\ f\,p \leq p\}.$$

Proof just like the version for sets.

Ordering $'a\ acom$

Any ordering on $'a$ can be lifted to $'a\ acom$ by comparing the annotations of $strip$-equal commands:

$$SKIP\ \{P\} \le SKIP\ \{P'\} \longleftrightarrow P \le P'$$

$$x ::= e\ \{P\} \le x' ::= e'\ \{P'\} \longleftrightarrow$$
$$x = x' \wedge e = e' \wedge P \le P'$$

$$C_1;\ C_2 \le C_1';\ C_2' \longleftrightarrow C_1 \le C_1' \wedge C_2 \le C_2'$$

$\vdots$

Ordering $'a\ acom$

For all other (not *strip*-equal) commands:

$$c \leq c' \longleftrightarrow \mathit{False}$$

Example:

$$\begin{array}{lll} x ::= N\ 0\ \{\{a\}\} \leq x ::= N\ 0\ \{\{a,\ b\}\} & \longleftrightarrow & \mathit{True} \\ x ::= N\ 0\ \{\{a\}\} \leq x ::= N\ 0\ \{\{\}\} & \longleftrightarrow & \mathit{False} \\ x ::= N\ 0\ \{S\} \leq x ::= N\ 1\ \{S\} & \longleftrightarrow & \mathit{False} \end{array}$$

The collecting semantics needs to order $state\ set\ acom$.

Annotations are (state) sets ordered by $\subseteq$,
which form a complete lattice.

Does $state\ set\ acom$ also form a complete lattice?

Almost . . .

A complication

What is the infimum of $SKIP \{S\}$ and $SKIP \{T\}$?

$$SKIP \{S \cap T\}$$

What is the infimum of $SKIP \{S\}$ and $x ::= N\ 0\ \{T\}$?

Only $strip$-equal commands have an infimum

It turns out:

- if $'a$ is a complete lattice,
- then for each $c :: com$
- the set $\{C :: 'a\ acom.\ strip\ C = c\}$
  is also a complete lattice
- but the whole type $'a\ acom$ is not.

Therefore we make the carrier set explicit.

Complete lattice as a set

Definition Let $'a$ be a partially ordered type.
A set $L :: 'a\ set$ is a *complete lattice*
if every $M \subseteq L$ has a greatest lower bound $\bigsqcap M \in L$.

Given sets A and B and a function f,
$f \in A \rightarrow B$ means $\forall\, a{\in}A.\ f\ a \in B$.

Theorem (Knaster-Tarski)
Let $L :: 'a\ set$ be a complete lattice
and $f \in L \rightarrow L$ a monotone function.
Then f (restricted to L) has the least fixed point

$$lfp\ f = \bigsqcap \{p \in L.\ f\ p \leq p\}.$$

Application to $acom$

Let $'a$ be a complete lattice and $c :: com$.
Then $L = \{C :: 'a\ acom.\ strip\ C = c\}$
is a complete lattice.

The infimum of a set $M \subseteq L$ is computed "pointwise":

Annotate c at annotation point p with the infimum of the annotations of all $C \in M$ at p.

Example $\bigsqcap \{SKIP\ \{A\}, SKIP\ \{B\}, \ldots\ \}$
$= SKIP\ \{\bigsqcap\ \{A,B, \ldots\}\}$

Formally ...

Some auxiliary functions:

Selecting subcommands:

$$sub_1\ (C_1;\ C_2) = C_1$$
$$sub_1\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) = C_1$$
$$sub_1\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = C$$

$$sub_2\ (C_1;\ C_2) = C_2$$
$$sub_2\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) = C_2$$

Selecting annotations:

$$anno_1\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) = P_1$$
$$anno_1\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = I$$

$$anno_2\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) = P_2$$
$$anno_2\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = P$$

The image of a set A under a function f:

$$f \text{ ` } A = \{y.\ \exists\, x{\in}A.\ y = f\, x\}$$

Predefined in HOL.

The union of $strip$-equal $acom$s:

$Union\_acom :: com \Rightarrow 'a\ acom\ set \Rightarrow 'a\ set\ acom$:

$$Union\_acom\ SKIP\ M \quad = \quad SKIP\ \{post\ `\ M\}$$

$$Union\_acom\ (x ::= a)\ M \quad = \quad x ::= a\ \{post\ `\ M\}$$

$$Union\_acom\ (c_1;\ c_2)\ M \quad =$$

$$Union\_acom\ c_1\ (sub_1\ `\ M);\ Union\_acom\ c_2\ (sub_2\ `\ M)$$

$$Union\_acom\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ M =$$
$$IF\ b\ THEN\ \{anno_1\ `\ M\}\ Union\_acom\ c_1\ (sub_1\ `\ M)$$
$$ELSE\ \{anno_2\ `\ M\}\ Union\_acom\ c_2\ (sub_2\ `\ M)$$
$$\{post\ `\ M\}$$

$$Union\_acom\ (WHILE\ b\ DO\ c)\ M =$$
$$\{anno_1\ `\ M\}$$
$$WHILE\ b$$
$$DO\ \{anno_2\ `\ M\}$$
$$\quad Union\_acom\ c\ (sub_1\ `\ M)$$
$$\{post\ `\ M\}$$

Lemma Let $'a$ be a complete lattice and $c :: com$.
Then $L = \{C :: 'a\ acom.\ strip\ C = c\}$
is a complete lattice where the infimum of $M \subseteq L$ is

$$map\_acom \bigsqcap (Union\_acom\ c\ M)$$

Proof of the infimum properties by induction on c.

The Collecting Semantics

The underlying complete lattice is now $state\ set$.

Therefore $L = \{C :: state\ set\ acom.\ strip\ C = c\}$ is a complete lattice for any c.

Lemma $step\ S \in L \to L$ and is monotone.

Therefore Knaster-Tarski is applicable and we define

$$CS :: com \Rightarrow state\ set\ acom$$
$$CS\ c = lfp\ c\ (step\ UNIV)$$

[lfp is defined in the context of some lattice L.
Our concrete L depends on c.
Therefore lfp depends on c, too.]

Relationship to big-step semantics

For simplicity: compare only pre and post-states

Theorem $(c,\ s) \Rightarrow t \Longrightarrow t \in post\ (CS\ c)$

Follows directly from

$$[\![\ (c,\ s) \Rightarrow t;\ s \in S\ ]\!] \Longrightarrow t \in post(lfp\ c\ (step\ S))$$

Proof of

$$\llbracket\ (c,\ s) \Rightarrow t;\ s \in S\ \rrbracket \Longrightarrow t \in post(lfp\ c\ (step\ S))$$

uses

$$post(lfp\ c\ f) = \bigcap\{post\ C\,|\,C.\ strip\ C = c \wedge f\ C \leq C\}$$

and

$$\llbracket(c,\ s) \Rightarrow t;\ strip\ C = c;\ s \in S;\ step\ S\ C \leq C\rrbracket \Longrightarrow t \in post\ C$$

which is proved by induction on the big step.

In a nutshell:
collecting semantics overapproximates big-step semantics

Later:
program analysis overapproximates collecting semantics
Together:
program analysis overapproximates big-step semantics

The other direction

$$t \in post(lfp\ c\ (step\ S)) \Longrightarrow \exists\, s{\in}S.\ (c,s) \Rightarrow\ t$$

is also true but is not proved in this course.

Approximating the Collecting semantics

A conceptual step:

$$(vname \Rightarrow val)\ set \quad \rightsquigarrow \quad vname \Rightarrow val\ set$$

A domain-specific step:

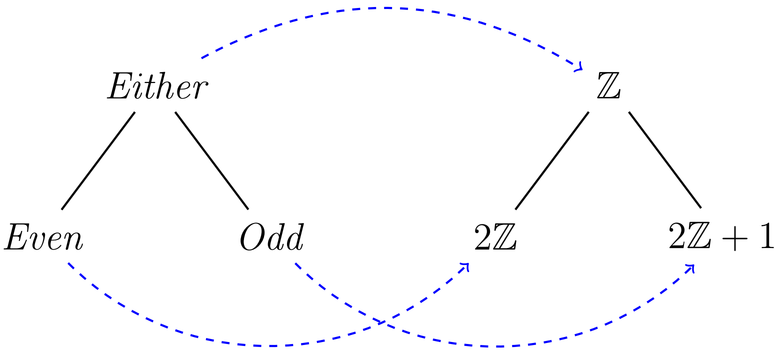
$$val\ set \quad \rightsquigarrow \quad 'av$$

where $'av$ is some ordered type of abstract values that we can compute on.

Example: parity analysis

Abstract values:

datatype $parity = Even \mid Odd \mid Either$



concretization function γ_{parity}

A *concretisation function* γ
maps an abstract value to a set of concrete values

Bigger abstract values represent more concrete values

Preorder

A type $'a$ is a *preorder* if

- there is a predicate $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow bool$
- that is *reflexive* ($x \sqsubseteq x$) and
- *transitive* ($[\![x \sqsubseteq y;\ y \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq z$)

A *partial order* is also *antisymmetric*
($[\![x \sqsubseteq y;\ y \sqsubseteq x]\!] \Longrightarrow x = y$)

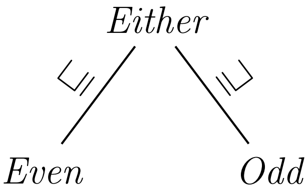
Pre vs partial

Partial orders are technically simpler.

Preorders are more liberal:

- they allow different representations for the same abstract element.
  Example: the intervals $[1, 0]$ and $[2, 0]$ both represent the empty interval.
- Instead of $x = y$, test for $x \sqsubseteq y \wedge y \sqsubseteq x$.

Example: parity



Fact Type $parity$ is a partial order.

Semilattice

A type $'a$ is a *semilattice* (with top element) if

- it is a preorder and
- there is a least upper bound operation
  $\sqcup :: 'a \Rightarrow 'a \Rightarrow 'a$

  $x \sqsubseteq x \sqcup y \qquad y \sqsubseteq x \sqcup y$

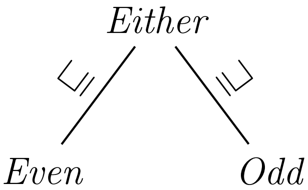
  $[\![x \sqsubseteq z;\ y \sqsubseteq z]\!] \Longrightarrow x \sqcup y \sqsubseteq z$
- and a top element $\top :: 'a$
  $x \sqsubseteq \top$

Application: abstract $\cup$, join two computation paths
We often call $\sqcup$ the *join* operation.

Lemma If $'a$ is a semilattice where $\sqsubseteq$ is actually a partial order, then the least upper bound of two elements is uniquely determined (and similarly the top element).

$\sqsubseteq$ uniquely determines $\sqcup$ and $\top$

Example: parity



Fact Type *parity* is a semilattice with top element.

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Examples	class $preord$:	preorders
	class $semilattice$:	semilattices

A type belongs to some class if

- the interface functions are defined on that type
- and satisfy the axioms of the class (proof needed!)

Notation: $\tau :: C$ means type τ belongs to class C

Example: $parity :: semilattice$

Abs_Int0.thy
Abs_Int1_parity.thy

Orderings

From abstract values to abstract states

Need to abstract collecting semantics:

$$state\ set$$

- First attempt:

  $$'av\ st\ =\ vname \Rightarrow 'av$$

  where $'av$ is the type of abstract values
- Problem: cannot abstract empty set of states (unreachable program points!)
- Solution: type $'av\ st\ option$

Lifting semilattice and γ to $'av\ st\ option$

Lemma If $'a :: semilattice$ then $'b \Rightarrow 'a :: semilattice$.
Proof
$(f \sqsubseteq g) = (\forall x.\ f\ x \sqsubseteq g\ x)$
$f \sqcup g = (\lambda x.\ f\ x \sqcup g\ x)$
$\top = (\lambda x.\ \top)$

definition
$\gamma\_fun :: ('a \Rightarrow 'c\ set) \Rightarrow ('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'c) set$
where $\gamma\_fun\ \gamma\ F = \{f.\ \forall x.\ f\ x \in \gamma\ (F\ x)\}$

Lemma If γ is monotone then $\gamma\_fun\ \gamma$ is monotone.

Lemma

If $'a :: semilattice$ then $'a\ option :: semilattice$.

Proof

$$(Some\ x \sqsubseteq Some\ y) = (x \sqsubseteq y)$$
$$(None \sqsubseteq \_) = True$$
$$(Some\ \_ \sqsubseteq None) = False$$

$$Some\ x \sqcup Some\ y = Some\ (x \sqcup y)$$
$$None \sqcup y = y$$
$$x \sqcup None = x$$

$$\top = Some\ \top$$

Corollary

If $'a :: semilattice$ then $'a\ st\ option :: semilattice$.

fun $\gamma\_option :: ('a \Rightarrow 'c\ set) \Rightarrow 'a\ option \Rightarrow 'c\ set$
where
$\gamma\_option\ \gamma\ None = \{\}$
$\gamma\_option\ \gamma\ (Some\ a) = \gamma\ a$

Lemma If γ is monotone then $\gamma\_option\ \gamma$ is monotone.

$'a\ acom$

Lemma If $'a :: preord$ then $'a\ acom :: preord$.
Proof $\sqsubseteq$ is lifted from $'a$ to $'a\ acom$ just like $\leq$.

Preorder is enough, semilattice not needed.

Lifting $\gamma :: 'a \Rightarrow 'c$ to $'a\ acom \Rightarrow 'c\ acom$ is easy:
map_acom

Lemma If γ is monotone then $map\_acom\ \gamma$ is monotone.

- Stepwise development of a
  generic abstract interpreter
  as a parameterized module
- Parameters/Input: abstract type of values together with abstractions of the operations on concrete type $val = int$.
- Result/Output: abstract interpreter that approximates the collecting semantics by computing on abstract values.
- Realization in Isabelle as a *locale*

Parameters (I)

Abstract values: type $'av :: semilattice$

Concretization function: $\gamma :: 'av \Rightarrow val\ set$

Assumptions: $a \sqsubseteq b \Longrightarrow \gamma\ a \subseteq \gamma\ b$

$\gamma\ \top = UNIV$

Parameters (II)

Abstract arithmetic: $num' :: val \Rightarrow 'av$
$plus' :: 'av \Rightarrow 'av \Rightarrow 'av$

Intention: num' abstracts the meaning of N
$plus'$ abstracts the meaning of $Plus$

Required for each constructor of $aexp$ (except V)

Assumptions:

$i \in \gamma\ (num'\ i)$

$[\![i_1 \in \gamma\ a_1;\ i_2 \in \gamma\ a_2]\!] \Longrightarrow i_1 + i_2 \in \gamma\ (plus'\ a_1\ a_2)$

The $n \in \gamma\ a$ relationship is maintained

Lifted concretization functions

$\gamma_s :: {}'av\ st \Rightarrow state\ set$
$\gamma_s = \gamma\_fun\ \gamma$

$\gamma_o :: {}'av\ st\ option \Rightarrow state\ set$
$\gamma_o = \gamma\_option\ \gamma_s$

$\gamma_c :: {}'a\ st\ option\ acom \Rightarrow state\ set\ acom$
$\gamma_c\ c = map\_acom\ \gamma_o\ c$

All of them are monotone.

Abstract interpretation of $aexp$

fun $aval' :: aexp \Rightarrow {'av}\ st \Rightarrow {'av}$
$aval'\ (N\ n)\ S = num'\ n$
$aval'\ (V\ x)\ S = S\ x$
$aval'\ (Plus\ a_1\ a_2)\ S = plus'\ (aval'\ a_1\ S)\ (aval'\ a_2\ S)$

Correctness of $aval'$ wrt $aval$:

Lemma $s \in \gamma_s\ S \Longrightarrow aval\ a\ s \in \gamma\ (aval'\ a\ S)$

Proof by induction on a
using the assumptions about the parameters.

Example instantiation with $parity$

$\sqsubseteq / \sqcup$ and γ_{parity}: see earlier

$$num\_parity\ i = (\textit{if}\ i\ mod\ 2 = 0\ \textsf{then}\ Even\ \textsf{else}\ Odd)$$

$$\begin{aligned}
&plus_parity\ Even\ Even = Even\\
&plus_parity\ Odd\ Odd = Even\\
&plus_parity\ Even\ Odd = Odd\\
&plus_parity\ Odd\ Even = Odd\\
&plus_parity\ Either\ y = Either\\
&plus_parity\ x\ Either = Either
\end{aligned}$$

Example instantiation with $parity$

Input:

$$\begin{array}{lcl} \gamma & \mapsto & \gamma_{parity} \\ num' & \mapsto & num\_parity \\ plus' & \mapsto & plus\_parity \end{array}$$

Must prove parameter assumptions

Output: $aval' \mapsto aval\_parity$

Example The value of

$$aval\_parity\ (Plus\ (V\ ''x'')\ (V\ ''x''))\\ \quad ((\lambda\_.\ Either)(''x'' := Odd))$$

is $Even$.

Abs_Int1_parity.thy

Locale interpretation

Abstract interpretation of *bexp*

For now, boolean expressions are not analysed.

Abstract interpretation of com

Abstracting the collecting semantics

$$step :: \ \tau \Rightarrow \tau \ \ acom \Rightarrow \tau \ \ acom$$
$$\text{where} \ \ \tau = state \ set$$

to

$$step' :: \ \tau \Rightarrow \tau \ \ acom \Rightarrow \tau \ \ acom$$
$$\text{where} \ \ \tau = {}'av \ st \ option$$

$$step'\ S\ (SKIP\ \{\_\}) = SKIP\ \{S\}$$

$$step'\ S\ (x ::= e\ \{\_\}) =$$
$$x ::= e$$
$$\{\textsf{case}\ S\ \textsf{of}\ None \Rightarrow None$$
$$\mid Some\ S \Rightarrow Some\ (S(x := aval'\ e\ S))\}$$

$$step'\ S\ (C_1;\ C_2) = step'\ S\ C_1;\ step'\ (post\ C_1)\ C_2$$

$$step'\ S\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{\_\}) =$$
$$IF\ b\ THEN\ \{S\}\ step'\ P_1\ C_1$$
$$ELSE\ \{S\}\ step'\ P_2\ C_2$$
$$\{post\ C_1 \sqcup post\ C_2\}$$

$$step'\ S\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{\_\}) =$$
$$\{S \sqcup post\ C\}\ WHILE\ b\ DO\ \{I\}\ step'\ P\ C\ \{I\}$$

Example: iterating $step\_parity$

$$(step\_parity\ S)^k\ c$$

where

$$\begin{aligned}
c = \quad & x ::= N\ 3\ \{None\}\ ;\\
& \{None\}\\
& WHILE\ b\ DO\ \{None\}\\
& \quad x ::= Plus\ (V\ x)\ (N\ 5)\ \{None\}\\
& \{None\}\\
S = \quad & Some\ (\lambda_.\ Either)\\
S_p = \quad & Some\ ((\lambda_.\ Either)(x := p))
\end{aligned}$$

Correctness of $step'$ wrt $step$

The conretization of $step'$ overaproximates $step$:

Lemma $step\ (\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ S\ C)$

where $S :: {}'av\ st\ option$
$C :: {}'av\ st\ option\ acom$

Proof by an easy induction on C

The abstract interpreter

- Ideally: iterate $step'$ until a fixed point is reached
- May take too long
- Sufficient: any post-fixed point: $step'\ S\ C \sqsubseteq C$
  Means iteration does not increase annotations,
  i.e. annotations are consistent but maybe too big
- Also remember: $\sqsubseteq$ only preorder, $=$ too strong

Unbounded search

From the HOL library:

$while\_option ::$
$\quad ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$

such that

$while\_option\ b\ f\ x =$
$(\textit{if}\ b\ x\ \textit{then}\ while\_option\ b\ f\ (f\ x)\ \textit{else}\ Some\ x)$

and $while\_option\ b\ f\ x = None$
if the recursion does not terminate.

Post-fixed point:

$pfp :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$

$pfp\ f = while\_option\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ f$

Start iteration with least annotated command:

$bot\ c = anno\ None\ c$

A transfer lemma

If $\sqsubseteq$ :: $'a \Rightarrow 'a \Rightarrow bool$
$\leq$ :: $'c \Rightarrow 'c \Rightarrow bool$ (transitive)
f' :: $'a \Rightarrow 'a$
f :: $'c \Rightarrow 'c$
g :: $'a \Rightarrow 'c$ (monotone)
$f\ (g\ x) \leq g\ (f'\ x)$ for all x :: $'a$

then, if p is a pfp of f', $g\ p$ is a pfp of f.

Proof $f\ (g\ p) \leq g\ (f'\ p)$ and $g\ (f'\ p) \leq g\ p$ because $f'\ p \sqsubseteq p$ and g is monotone.

The generic abstract interpreter

definition $AI :: com \Rightarrow {}'av\ st\ option\ acom\ option$
where $AI\ c = pfp\ (step'\ \top)\ (bot\ c)$

Theorem $AI\ c = Some\ C \Longrightarrow CS\ c \le \gamma_c\ C$

Proof From the assumption: C is a pfp of $step'\ \top$.
Because of the correctness of $step'$ wrt $step$,
monotonicity of γ_c and the transfer lemma:
$\gamma_c\ C$ is a pfp of $step\ (\gamma_o\ \top) = step\ UNIV$.
Because CS is the least pfp of $step\ UNIV$: $CS\ c \le \gamma_c\ C$.

Problem

AI is not directly executable

because pfp compares $f\ C \sqsubseteq C$
where $C :: {'av}\ st\ option\ acom$
which compares functions $vname \Rightarrow {'av}$
which is (in general) uncomputable: $vname$ is infinite.

Solution

Program states are finite functions
from the variables actually present in a program.

Thus we replace $'av\ st = vname \Rightarrow 'av$ by

datatype $'av\ st =$
$FunDom\ (vname \Rightarrow 'av)\ (vname\ set)$

where $FunDom\ f\ X$ represents a function f with an explicit domain X. In our application X will be finite.

Projections: $fun\ (FunDom\ f\ X) = f$

$dom\ (FunDom\ f\ X) = X$

Update:

$update\ F\ x\ y = FunDom\ ((fun\ F)(x := y))\ (dom\ F)$

Application: domain stays fixed

Concretization:

$\gamma_s\ F = \{f.\ \forall\, x \in dom\ F.\ f\ x \in \gamma\ (fun\ F\ x)\}$

Making $'a\ st$ a semilattice

Assuming $'a$ is a semillatice.
Natural ordering on $'a\ st$:

$(F \sqsubseteq G) =$
$(dom\ F = dom\ G \wedge (\forall\, x{\in}dom\ F.\ fun\ F\ x \sqsubseteq fun\ G\ x))$

Does not make all of $'a\ st$ a semilattice,
$F \sqcup G$ exists only if $dom\ F = dom\ G$.

Generic solution: refine definition of semilattice with explicit carrier set L (like complete lattice earlier).

This time we make the dependence of L on the context explicit.

The context is the set of variables X in the program.

Now $x \sqsubseteq x \sqcup y$ becomes

$$[\![x \in L\ X;\ y \in L\ X]\!] \Longrightarrow x \sqsubseteq x \sqcup y$$

Semilattice with carrier set

- $L :: vname\ set \Rightarrow {}'a\ set$
  $\sqcup :: {}'a \Rightarrow {}'a \Rightarrow {}'a$
  $\top :: char\ list\ set \Rightarrow {}'a$
- $[\![x \in L\ X;\ y \in L\ X]\!] \Longrightarrow x \sqsubseteq x \sqcup y$
  $[\![x \in L\ X;\ y \in L\ X]\!] \Longrightarrow y \sqsubseteq x \sqcup y$
  $[\![x \sqsubseteq z;\ y \sqsubseteq z]\!] \Longrightarrow x \sqcup y \sqsubseteq z$

  $x \in L\ X \Longrightarrow x \sqsubseteq \top_X$

  $[\![x \in L\ X;\ y \in L\ X]\!] \Longrightarrow x \sqcup y \in L\ X$
  $\top_X \in L\ X$

Isabelle class: *semilatticeL*

Type st as a semilattice

Lemma If $'a :: semilattice$ then $'a\ st :: semilatticeL$.

Proof

$L\ X = \{F.\ dom\ F = X\}$

$(F \sqsubseteq G) =$
$(dom\ F = dom\ G \wedge (\forall x{\in}dom\ F.\ fun\ F\ x \sqsubseteq fun\ G\ x))$

$F \sqcup G = FunDom\ (\lambda x.\ fun\ F\ x \sqcup fun\ G\ x)\ (dom\ F)$

$\top_X = FunDom\ (\lambda x.\ \top)\ X$

Type $option$ as a semilattice

Lemma

If $'a :: semilatticeL$ then $'a\ option :: semilatticeL$.

Proof

$None \in L\ X$

$(Some\ x \in L\ X) = (x \in L\ X)$

Operations $\sqsubseteq$ and $\sqcup$: see earlier

Generic abstract interpreter

Everything as before, except

- new definition of st
- for $S :: {}'av\ st$:

$$\begin{array}{lcl} S\ x & \leadsto & fun\ S\ x \\ S(x := a) & \leadsto & update\ S\ x\ a \end{array}$$

- $AI\ c = pfp\ (step'\ \top)\ (bot\ c)\ \leadsto$
  $AI\ c = pfp\ (step'\ \top_c)\ (bot\ c)$

The abstract interpreter is computable

because all the abstract states during its computation have the finite domain $vars\ c$

because the computation starts with $\top_c :: {'av}\ st$ and does not involve variables outside $vars\ c$.

Abs_Int1_parity.thy
Abs_Int1_const.thy

Examples

Beyond partial correctness

- AI may compute any pfp
- AI may not terminate

The solution: Monotonicity

$\Longrightarrow$

Precision AI computes *least* post-fixed points

Termination AI terminates if $'av$ is of bounded height

Monotonicity

The *monotone framework* also demands monotonicity of abstract arithmetic:

$$[\![a_1 \sqsubseteq b_1;\ a_2 \sqsubseteq b_2]\!] \Longrightarrow plus'\ a_1\ a_2 \sqsubseteq plus'\ b_1\ b_2$$

Theorem In the monotone framework, $aval'$ is also monotone

$$S_1 \sqsubseteq S_2 \Longrightarrow aval'\ e\ S_1 \sqsubseteq aval'\ e\ S_2$$

if $S_1 \in L\ X$, $S_2 \in L\ X$, $vars\ e \subseteq X$

and therefore $step'$ is also monotone:

$$[\![S_1 \sqsubseteq S_2;\ C_1 \sqsubseteq C_2]\!] \Longrightarrow step'\ S_1\ C_1 \sqsubseteq step'\ S_2\ C_2$$

if $S_1 \in L\ X$, $S_2 \in L\ X$, $C_1 \in L\ X$, $C_2 \in L\ X$

Precision: smaller is better

If f is monotone and $\bot$ is a least element,
then $pfp\ f\ \bot$ is a least post-fixed point of f

Lemma Let $\sqsubseteq$ be a preorder on a set L with least element $\bot \in L$: $x \in L \Longrightarrow \bot \sqsubseteq x$.
Let $f \in L \to L$ be a monotone function: $x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$.
If $\mathit{while\_option}\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ f\ \bot = \mathit{Some}\ p$ then p is a least post-fixed point of f.
That is, if $f\ q \sqsubseteq q$ for some $q \in L$, then $p \sqsubseteq q$.

Proof Clearly $f\ p \sqsubseteq p$. Given any post-fixed point $q \in L$, property $P\ x = (x \in L \wedge x \sqsubseteq q)$ is an invariant of the while loop: $P\ \bot$ holds and $P\ x$ implies $f\ x \sqsubseteq f\ q \sqsubseteq q$. Hence upon termination, $P\ p$ must also hold and hence $p \sqsubseteq q$.

Application to

$$AI\ c = pfp\ (step'\ \top_{vars\ c})\ (bot\ c)$$
$$pfp\ f = while\_option\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ f$$

Because $bot\ c$ is a least element and $step'$ is monotone, AI returns least post-fixed points

Termination

Definition $x \sqsubset y \longleftrightarrow x \sqsubseteq y \wedge \neg\, y \sqsubseteq x$

Because $step'$ is monotone, starting from $bot\ c$ generates an ascending $\sqsubset$ chain of annotated commands.
We exhibit a measure function m_c that decreases with every loop iteration:

$$C_1 \sqsubset C_2 \Longrightarrow m_c\ C_2 < m_c\ C_1$$

The measure function m_c is constructed from a measure function m on $'av$ in several steps.

Parameters: $m :: 'av \Rightarrow nat$
$h :: nat$

Assumptions: $m\ x \leq h$
$x \sqsubseteq y \Longrightarrow m\ y \leq m\ x$
$x \sqsubset y \Longrightarrow m\ y < m\ x$

Parameter h is the height of $\sqsubset$: every ascending chain $x_0 \sqsubset x_1 \sqsubset \ldots$ has length at most h.

Application to *parity* and *const*: $h = 1$

Lifting m to abstract states:

$m_s :: {'av}\ st \Rightarrow nat$

$m_s\ S = (\sum x{\in}dom\ S.\ m\ (fun\ S\ x))$

Lemmas

$m_s\ x \leq h * card\ X$ if $x \in L\ X$, $finite\ X$

$S_1 \sqsubseteq S_2 \Longrightarrow m_s\ S_2 \leq m_s\ S_1$

$S_1 \sqsubset S_2 \Longrightarrow m_s\ S_2 < m_s\ S_1$ if $finite\ (dom\ S_1)$

Lifting m_s to options:

$m_o :: nat \Rightarrow {'av}\ st\ option \Rightarrow nat$

$m_o\ d\ (Some\ S) = m_s\ S$

$m_o\ d\ None = h * d + 1$

Lemmas

$m_o\ (card\ X)\ ost \leq h * card\ X + 1$

if $ost \in L\ X$, $finite\ X$

$o_1 \sqsubseteq o_2 \Longrightarrow m_o\ (card\ X)\ o_2 \leq m_o\ (card\ X)\ o_1$

if $finite\ X$, $o_1 \in L\ X$, $o_2 \in L\ X$

$o_1 \sqsubset o_2 \Longrightarrow m_o\ (card\ X)\ o_2 < m_o\ (card\ X)\ o_1$

if $finite\ X$, $o_1 \in L\ X$, $o_2 \in L\ X$

Lifting m_o to annotated commands:

m_c :: $'av\ st\ option\ acom \Rightarrow nat$

$$m_c\ C =$$
$$(\sum i{<}length\ (annos\ C).$$
$$m_o\ (card\ (vars\ (strip\ C)))\ (annos\ C\ !\ i))$$

Theorems

$$m_c\ C$$
$$\leq length\ (annos\ C) *$$
$$(h * card\ (vars\ (strip\ C)) + 1)$$

if $C \in L\ (vars\ (strip\ C))$

$$C_1 \sqsubset C_2 \Longrightarrow m_c\ C_2 < m_c\ C_1$$

if $C_1 \in L\ (vars\ (strip\ C_1))$, $C_2 \in L\ (vars\ (strip\ C_2))$

Thus we have not only proved termination but also complexity:

$$AI\ c \text{ needs at most } p*(n*h+1) \text{ steps}$$

where $p = $ number of annotation in c
$n = $ number of variables in c

Warning: $step'$ is very inefficient.
It is applied to every subcommand in every step.
Thus the actual complexity of AI is $O(p^2 * n * h)$

Better iteration policy:
Ignore subcommands where nothing has changed.

Practical algorithms often use a control flow graph and a worklist recording the nodes where annotations have changed.

As usual: efficiency complicates proofs.

Abs_Int1_parity.thy
Abs_Int1_const.thy

Termination

Need to simulate collecting semantics (S :: $state\ set$):

$$\{s \in S.\ bval\ b\ s\}$$

Given S :: $'av\ st$, reduce it to some $S' \sqsubseteq S$ such that

$$\text{if } s \in \gamma_s\ S \text{ and } bval\ b\ s \text{ then } s \in \gamma_s\ S'$$

- No state satisfying b is lost
- but $\gamma_s\ S'$ may still contain states not satisfying b.
- Trivial solution: $S' = S$

Computing S' from S requires $\sqcap$

Lattice

A type $'a$ is a *lattice* (with top and bottom) if

- it is a semilattice (with top)
- there is a greatest lower bound operation
  $\sqcap :: 'a \Rightarrow 'a \Rightarrow 'a$

  $x \sqcap y \sqsubseteq x \qquad x \sqcap y \sqsubseteq y$

  $[\![z \sqsubseteq x; z \sqsubseteq y]\!] \Longrightarrow z \sqsubseteq x \sqcap y$
- and a *bottom* element $\bot :: 'a$
  $\bot \sqsubseteq x$

We often call $\sqcap$ the *meet* operation.

Type class: $lattice$

Concretization

We strengthen the abstract interpretation framework by assuming

- $'av :: lattice$
- $\gamma\ a_1 \cap \gamma\ a_2 \subseteq \gamma\ (a_1 \sqcap a_2)$

  $\Longrightarrow \gamma\ (a_1 \sqcap a_2) = \gamma\ a_1 \cap \gamma\ a_2$
  $\Longrightarrow \sqcap$ is precise!

  How about $\gamma\ a_1 \cup \gamma\ a_2$ and $\gamma\ (a_1 \sqcup a_2)$?
- $\gamma\ \bot = \{\}$

Backward analysis of $aexp$

Given $e :: aexp$
$a :: 'av$ (the intended value of e)
$S :: 'av\ st$
restrict S to some $S' \sqsubseteq S$ such that

$$\{s \in \gamma_s\ S.\ aval\ e\ s \in \gamma\ a\} \subseteq \gamma_s\ S'$$

Roughly: S' overapproximates the subset of S that makes e evaluate to a.

What if $\{s \in \gamma_s\ S.\ aval\ e\ s \in \gamma\ a\}$ is empty?
Work with $'av\ st\ option$ instead of $'av\ st$

$afilter\ N$

$afilter :: aexp \Rightarrow 'av \Rightarrow 'av\ st\ option \Rightarrow 'av\ st\ option$

$afilter\ (N\ n)\ a\ S = (\textbf{if}\ test\_num'\ n\ a\ \textbf{then}\ S\ \textbf{else}\ None)$

An extension of the interface of our framework:

$test\_num' :: int \Rightarrow 'av \Rightarrow bool$

Assumption:

$test\_num'\ n\ a = (n \in \gamma\ a)$

Needed only for computability reasons.

$afilter\ V$

$afilter\ (V\ x)\ a\ S =$

case S *of* $None \Rightarrow None$
$|\ Some\ S \Rightarrow$
  let $a' = fun\ S\ x \sqcap a$
  in if $a' \sqsubseteq \bot$ *then* $None$
    else $Some\ (update\ S\ x\ a')$

Avoid $\bot$ component in abstract state,
turn abstract state into $None$ instead.

$\mathit{afilter}\ \mathit{Plus}$

A further extension of the interface of our framework:

$$\mathit{filter\_plus}' :: {}'av \Rightarrow {}'av \Rightarrow {}'av \Rightarrow {}'av \times {}'av$$

Assumption:

$$\begin{aligned}&\mathit{filter\_plus}'\ a\ a_1\ a_2 = (a_1',\ a_2') \Longrightarrow\\ &\quad \gamma\ a_1' \supseteq \{i_1 \in \gamma\ a_1.\ \exists\, i_2 \in \gamma\ a_2.\ i_1{+}i_2 \in \gamma\ a\} \wedge\\ &\quad \gamma\ a_2' \supseteq \{i_2 \in \gamma\ a_2.\ \exists\, i_1 \in \gamma\ a_1.\ i_1{+}i_2 \in \gamma\ a\}\end{aligned}$$

Definition:

$$\begin{aligned}&\mathit{afilter}\ (\mathit{Plus}\ e_1\ e_2)\ a\ S =\\ &(\textsf{let}\ (a_1,\ a_2) = \mathit{filter\_plus}'\ a\ (\mathit{aval}''\ e_1\ S)\ (\mathit{aval}''\ e_2\ S)\\ &\ \textsf{in}\ \mathit{afilter}\ e_1\ a_1\ (\mathit{afilter}\ e_2\ a_2\ S))\end{aligned}$$

(Analogously for all other arithmetic operations)

Backward analysis of $bexp$

Given $b :: bexp$
$res :: bool$ (the intended value of b)
$S :: 'av\ st\ option$
restrict S to some $S' \sqsubseteq S$ such that

$$\{s \in \gamma_o\ S.\ bval\ b\ s = res\} \subseteq \gamma_o\ S'$$

Roughly: S' overapproximates the subset of S that makes b evaluate to res.

$$bfilter :: bexp \Rightarrow bool \Rightarrow {}'av\ st\ option \Rightarrow {}'av\ st\ option$$

$$bfilter\ (Bc\ v)\ res\ S = (\textsf{if}\ v = res\ \textsf{then}\ S\ \textsf{else}\ None)$$

$$bfilter\ (Not\ b)\ res\ S = bfilter\ b\ (\neg\ res)\ S$$

$$bfilter\ (And\ b_1\ b_2)\ res\ S =$$
$$\textsf{if}\ res\ \textsf{then}\ bfilter\ b_1\ True\ (bfilter\ b_2\ True\ S)$$
$$\textsf{else}\ bfilter\ b_1\ False\ S \sqcup bfilter\ b_2\ False\ S$$

$$bfilter\ (Less\ e_1\ e_2)\ res\ S =$$
$$\textsf{let}\ (a_1,\ a_2) = filter\_less'\ res\ (aval''\ e_1\ S)\ (aval''\ e_2\ S)$$
$$\textsf{in}\ afilter\ e_1\ a_1\ (afilter\ e_2\ a_2\ S)$$

A further extension of the interface of our framework:

$$\mathit{filter\_less}' :: \mathit{bool} \Rightarrow {}'\mathit{av} \Rightarrow {}'\mathit{av} \Rightarrow {}'\mathit{av} \times {}'\mathit{av}$$

Assumption:

$$\begin{aligned}
&\mathit{filter_less}'\ \mathit{res}\ a_1\ \ a_2 = (a_1',\ a_2') \Longrightarrow \\
&\quad \gamma\ \ a_1' \supseteq \{i_1 \in \gamma\ \ a_1.\ \exists\, i_2 \in \gamma\ \ a_2.\ (i_1 {<} i_2) = \mathit{res}\} \wedge \\
&\quad \gamma\ \ a_2' \supseteq \{i_2 \in \gamma\ \ a_2.\ \exists\, i_1 \in \gamma\ \ a_1.\ (i_1 {<} i_2) = \mathit{res}\}
\end{aligned}$$

$step'$

$step'\ S\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$
$IF\ b\ THEN\ \{bfilter\ b\ True\ S\}\ step'\ P_1\ C_1$
$ELSE\ \{bfilter\ b\ False\ S\}\ step'\ P_2\ C_2$
$\{post\ C_1 \sqcup post\ C_2\}$

$step'\ S\ (\{I\}\ WHILE\ b\ DO\ \{p\}\ C\ \{Q\}) =$
$\{S \sqcup post\ C\}$
$WHILE\ b$
$DO\ \{bfilter\ b\ True\ I\}$
$\quad step'\ p\ C$
$\{bfilter\ b\ False\ I\}$

Correctness proof

Almost as before, but with correctness lemmas for $afilter$

$\{s \in \gamma_o\ S.\ aval\ e\ s \in \gamma\ a\} \subseteq \gamma_o\ (afilter\ e\ a\ S)$
if $S \in L\ X$, $vars\ e \subseteq X$

and $bfilter$:

$\{s \in \gamma_o\ S.\ bv = bval\ b\ s\} \subseteq \gamma_o\ (bfilter\ b\ bv\ S)$
if $S \in L\ X$, $vars\ b \subseteq X$

Summary

Extended interface to abstract interpreter:

- $'av :: lattice$
  $\gamma \ \bot = \{\}$ and $\gamma \ a_1 \cap \gamma \ a_2 \subseteq \gamma \ (a_1 \sqcap a_2)$
- $test\_num' :: int \Rightarrow 'av \Rightarrow bool$
  $test\_num' \ n \ a = (n \in \gamma \ a)$
- $filter\_plus' :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$
  $[\![filter\_plus' \ a \ a_1 \ a_2 = (a_1', a_2');$
  $i_1 \in \gamma \ a_1; \ i_2 \in \gamma \ a_2; \ i_1 + i_2 \in \gamma \ a]\!]$
  $\Longrightarrow i_1 \in \gamma \ a_1' \wedge i_2 \in \gamma \ a_2'$
- $filter\_less' :: bool \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$
  $[\![filter\_less' \ (i_1 < i_2) \ a_1 \ a_2 = (a_1', a_2');$
  $i_1 \in \gamma \ a_1; \ i_2 \in \gamma \ a_2]\!]$
  $\Longrightarrow i_1 \in \gamma \ a_1' \wedge i_2 \in \gamma \ a_2'$

Abs_Int2_ivl.thy

The problem

If there are infinite ascending $\sqsubseteq$ chains of abstract values then the abstract interpreter may not terminate.

Canonical example: intervals

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq \ldots$$

Can happen even if the program terminates!

Widening

- $x_0 = \bot$, $x_{i+1} = f(x_i)$
  may not terminate while searching for a pfp:
  $f(x_i) \sqsubseteq x_i$
- Widen in each step: $x_{i+1} = x_i \triangledown f(x_i)$
  until a pfp is found.
- We assume
  - $\triangledown$ "extrapolates" its arguments: $x, y \sqsubseteq x \triangledown y$
  - $\triangledown$ "jumps" far enough to prevent nontermination

Example: Widening on intervals

$$[l_1,h_1] \bigtriangledown [l_2,h_2] = [l,h]$$

where
$$l = (\mathit{if}\ l_1 > l_2\ \mathit{then}\ -\infty\ \mathit{else}\ l_1)$$
$$h = (\mathit{if}\ h_1 < h_2\ \mathit{then}\ \ \infty\ \mathit{else}\ h_1)$$

Warning

- $x_{i+1} = f(x_i)$ finds a least pfp
  if it terminates, f is monotone, and $x_0 = \bot$
- $x_{i+1} = x_i \triangledown f(x_i)$ may return *any* pfp
  in the worst case $\top$

We win termination, we lose precision

A *widening operator* $\nabla :: 'a \Rightarrow 'a \Rightarrow 'a$ on a preorder must satisfy $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$.

Widening operators can be extended from $'a$ to $'a\ st$, $'a\ option$ and $'a\ acom$.

Abstract interpretation with widening

New assumption: $'av$ has widening operator

$iter\_widen :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$

$iter\_widen\ f =$
$while\_option\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ (\lambda x.\ x \nabla f\ x)$

Correctness (returns pfp): by definition

Abstract interpretation of c:

$$iter\_widen\ (step'\ \top_{vars\ c})\ (bot\ c)$$

Interval example

$x ::= N\ \mathit{0}\ \{A_0\};$
$\{A_1\}$
$\mathit{WHILE}\ \mathit{Less}\ (V\ x)\ (N\ \mathit{100})$
$\mathit{DO}\ \{A_2\}$
$\quad x ::= \mathit{Plus}\ (V\ x)\ (N\ \mathit{1})\ \{A_3\}$
$\{A_4\}$

Narrowing

Widening returns a (potentially) imprecise pfp p.

If f is monotone, further iteration improves p:

$$p \sqsupseteq f(p) \sqsupseteq f^2(p) \sqsupseteq \ldots$$

and each $f^i(p)$ is still a pfp!

- need not terminate: $[0, \infty] \sqsupset [1, \infty] \sqsupset \ldots$
- but we can stop at any point!

A *narrowing operator* $\triangle :: 'a \Rightarrow 'a \Rightarrow 'a$
must satisfy $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \triangle y \sqsubseteq x$.

Lemma Let f be monotone.
If $f\ p \sqsubseteq p$ then $f(p \triangle f\ p) \sqsubseteq p \triangle f\ p \sqsubseteq p$

$iter\_narrow\ f\ p =$
$while\_option\ (\lambda x.\ \neg\ x \sqsubseteq x \triangle f\ x)\ (\lambda x.\ x \triangle f\ x)\ p$

If f is monotone and p a pfp of f and the loop terminates, then (by the lemma) we obtain a pfp of f below p.

Iteration as long as progress is made: $x \triangle f\ x \sqsubset x$

Example: Narrowing on intervals

$$[l_1, h_1] \bigtriangleup [l_2, h_2] = [l, h]$$

where $l = (\mathit{if}\ l_1 = -\infty\ \mathit{then}\ l_2\ \mathit{else}\ l_1)$

$h = (\mathit{if}\ h_1 = \ \infty\ \mathit{then}\ h_2\ \mathit{else}\ h_1)$

Abstract interpretation with widening & narrowing

New assumption: $'av$ also has a narrowing operator

$pfp\_wn\ f\ x =$
$(\textsf{case}\ iter\_widen\ f\ x\ \textsf{of}\ None \Rightarrow None$
$\mid Some\ p \Rightarrow iter\_narrow\ f\ p)$

$AI\_wn\ c = pfp\_wn\ (step'\ \top_{vars\ c})\ (bot\ c)$

Theorem $AI\_wn\ c = Some\ C \Longrightarrow CS\ c \le \gamma_c\ C$
Proof as before

Termination

of

$$while\_option\ (\lambda x.\ P\ x)\ (\lambda x.\ g\ x)$$

via measure function m
such that m goes down with every iteration:

$$P\ x \Longrightarrow m\ x > m(g\ x)$$

May need some invariant Inv as additional premise:

$$Inv\ x \Longrightarrow P\ x \Longrightarrow m\ x > m(g\ x)$$

Termination of $iter\_widen$

$iter\_widen\ f =$
$while\_option\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ (\lambda x.\ x \bigtriangledown f\ x)$

As before: Assume $m :: 'av \Rightarrow nat$ such that $m\ x \leq h$
and $x \sqsubseteq y \Longrightarrow m\ y \leq m\ x$
but now $\neg\ y \sqsubseteq x \Longrightarrow m\ (x \bigtriangledown y) < m\ x$

Define the same functions $m_s/m_o/m_c$ on top.

Termination of $iter\_widen$ on $'a\ st\ option\ acom$:
Lemma $\neg\ C_2 \sqsubseteq C_1 \Longrightarrow m_c\ (C_1 \bigtriangledown C_2) < m_c\ C_1$
if $C_1 \in Lc\ c,\ C_2 \in Lc\ c$

Termination of $iter\_narrow$

$iter\_narrow\ f =$
$while\_option\ (\lambda x.\ \neg\ x \sqsubseteq x \bigtriangleup f\ x)\ (\lambda x.\ x \bigtriangleup f\ x)$

Assume $n :: 'av \Rightarrow nat$ such that

$x \sqsubseteq y \Longrightarrow n\ x \leq n\ y$
$[\![y \sqsubseteq x;\ \neg\ x \sqsubseteq x \bigtriangleup y]\!] \Longrightarrow n\ (x \bigtriangleup y) < n\ x$

Define $n_s/n_o/n_c$ like $m_s/m_o/m_c$

Termination of $iter\_narrow$ on $'a\ st\ option\ acom$:
Lemma $[\![\ C_2 \sqsubseteq C_1;\ \neg\ C_1 \sqsubseteq C_1 \bigtriangleup C_2\ ]\!] \Longrightarrow$
$n_c\ (C_1 \bigtriangleup C_2) < n_c\ C_1$ if $C_1 \in Lc\ c,\ C_2 \in Lc\ c$

Measuring intervals

$$\begin{aligned} m\ [l,h] = &\ (\mathit{if}\ l = -\infty\ \mathit{then}\ \mathit{0}\ \mathit{else}\ \mathit{1}) + \\ &\ (\mathit{if}\ h = \infty\ \mathit{then}\ \mathit{0}\ \mathit{else}\ \mathit{1}) \end{aligned}$$

$$h = \mathit{2}$$

$$n\ \mathit{ivl} = \mathit{2} - m\ \mathit{ivl}$$

Part VI

Extensions of IMP

25 Procedures and Local Variables

25 Procedures and Local Variables

25 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

New commands

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

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

Call procedure: $CALL\ p$

Concrete syntax

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

Abstract syntax

$$
\begin{array}{lcl}
\textbf{datatype}\ com & = & \ldots \text{basic commands} \ldots \\
 & | & \mathit{Var}\ \mathit{vname}\ \mathit{com} \\
 & | & \mathit{Proc}\ \mathit{pname}\ \mathit{com}\ \mathit{com} \\
 & | & \mathit{CALL}\ \mathit{pname}
\end{array}
$$

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

$$
\begin{array}{l}
\{VAR\ ''x'';;\\
\ \{PROC\ ''p'' = ''x'' ::= N\ 1;;\\
\ \ \{PROC\ ''q'' = CALL\ ''p'';;\\
\ \ \ \{VAR\ ''x'';;\ ''x'' ::= N\ 2;\\
\ \ \ \ \{PROC\ ''p'' = ''x'' ::= N\ 3;;\\
\ \ \ \ \ CALL\ ''q'';\ ''y'' ::= V\ ''x''\}\}\}\}\}
\end{array}
$$

What is the final value of variable y ?

- 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.

25 Procedures and Local Variables

Procedure environment

$$penv \;=\; pname \Rightarrow com$$

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.

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.

Record the static environment for each procedure together with the procedure body:

$$penv \;=\; pname \Rightarrow com \times penv$$

Recursive type synonyms not allowed.
Alternative: organize procedure environment like a stack.

$$penv \;=\; (pname \times com)\; list$$

The static environment of p is the $penv$ before $(p, \_)$ was added: pop until $(p, \_)$ is found.

Rules for new commands

$$\frac{pe \vdash (c,\ s) \Rightarrow t}{pe \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)}$$

$$\frac{(p,\ cp) \#\ pe \vdash (c,\ s) \Rightarrow t}{pe \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t}$$

$$\frac{(p,\ c) \#\ pe \vdash (c,\ s) \Rightarrow t}{(p,\ c) \#\ pe \vdash (CALL\ p,\ s) \Rightarrow t}$$

$$\frac{p' \neq p \qquad pe \vdash (CALL\ p,\ s) \Rightarrow t}{(p',\ c) \#\ pe \vdash (CALL\ p,\ s) \Rightarrow t}$$

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:

$$venv \ = \ vname \Rightarrow addr$$

A store associates addresses with values:

$$store \ = \ addr \Rightarrow val$$

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:

$$penv \;=\; (pname \times venv \times com)\; list$$

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

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

$$(pe,\ ve,\ f) \vdash (x ::= a,\ s) \Rightarrow s(ve\ x := aval\ a\ (s \circ ve))$$

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

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

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

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

$$\frac{bval\ b\ (s_1 \circ venv\ e) \quad e \vdash (c,\ s_1) \Rightarrow s_2 \quad e \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3}{e \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3}$$

Rules for new commands

$$\frac{(pe,\ ve(x := f),\ f + 1) \vdash (c,\ s) \Rightarrow t}{(pe,\ ve,\ f) \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t}$$

$$\frac{((p,\ cp,\ ve)\ \#\ pe,\ ve,\ f) \vdash (c,\ s) \Rightarrow t}{(pe,\ ve,\ f) \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t}$$

$$\frac{((p,\ c,\ ve)\ \#\ pe,\ ve,\ f) \vdash (c,\ s) \Rightarrow t}{((p,\ c,\ ve)\ \#\ pe,\ ve',\ f) \vdash (CALL\ p,\ s) \Rightarrow t}$$

$$\frac{p' \neq p \qquad (pe,\ ve,\ f) \vdash (CALL\ p,\ s) \Rightarrow t}{((p',\ c,\ ve')\ \#\ pe,\ ve,\ f) \vdash (CALL\ p,\ s) \Rightarrow t}$$