Concrete Semantics

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2023-1-31

Part II

Semantics

Chapter 7

IMP: A Simple Imperative Language

1. IMP Commands

2. Big-Step Semantics

3. Small-Step Semantics

1. IMP Commands

2. Big-Step Semantics

3. Small-Step Semantics

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:

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

Commands

Abstract syntax:

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

Com.thy

1. IMP Commands

2. Big-Step Semantics

3. Small-Step Semantics

Big-step semantics

Concrete syntax:

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

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

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

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

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

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*

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

(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

$$w \sim w'$$

where $\quad w = \mathit{WHILE}\ b\ \mathit{DO}\ c$

$\quad\quad\quad\; w' = \mathit{IF}\ b\ \mathit{THEN}\ c;;\ w\ \mathit{ELSE}\ \mathit{SKIP}$

Equivalence proof

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

$$\longleftrightarrow$$

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

$$\longleftrightarrow$$

$$(w',\ s) \Rightarrow t$$

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 $\nexists\, t.\ (c,\ s) \Rightarrow\ t$?

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

Big-step semantics cannot directly describe

- nonterminating computations,
- parallel computations.

We need a finer grained semantics!

1. IMP Commands

2. Big-Step Semantics

3. 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 $\nexists\, cs'.\ cs \to cs'$

The intention:

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

Why?

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

Small-step rules

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

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

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

Small-step rules

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

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

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

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

Small-step examples

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

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

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

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

Small_Step.thy

Semantics

Are big and small-step semantics equivalent?

From $\Rightarrow$ to $\rightarrow*$

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

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

In two cases a lemma is needed:

Lemma

$(c_1, s) \rightarrow* (c_1', s') \implies (c_1;; c_2, s) \rightarrow* (c_1';; c_2, s')$

Proof by rule induction.

From $\rightarrow*$ to $\Rightarrow$

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

Proof by rule induction on $cs \rightarrow* (SKIP, t)$.
In the induction step a lemma is needed:

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

Proof by rule induction on $cs \rightarrow cs'$.

Equivalence

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

Small_Step.thy

Equivalence of big and small

Can execution stop prematurely?

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

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

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

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

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

Together:

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

Infinite executions

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

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

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

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

(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 $cs \rightarrow cs' \rightarrow \ldots$

Chapter 8

Compiler

4. Stack Machine

5. Compiler

4 Stack Machine

5 Compiler

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:

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

Abbreviations:

$$hd2\ xs \equiv hd\ (tl\ xs)$$
$$tl2\ xs \equiv tl\ (tl\ xs)$$

Instruction execution

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

Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$\mathit{instr\ list} \vdash \mathit{config} \rightarrow \mathit{config}$$

$$0 \le i \wedge i < \mathit{size}\ P \Longrightarrow$$
$$P \vdash (i,\ s,\ \mathit{stk}) \rightarrow \mathit{iexec}\ (P\ !!\ i)\ (i,\ s,\ \mathit{stk})$$

where $'a\ \mathit{list}\ !!\ \mathit{int}$ = nth instruction of list
$\mathit{size} :: {'a}\ \mathit{list} \Rightarrow \mathit{int}$ = 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

4 Stack Machine

5 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:

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

Proof by induction on a (with arbitrary stk).

Needs lemmas!

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

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

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

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

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

Proofs by cases.

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

$bcomp\ b\ False\ 3 =$

$$\begin{array}{l}[LOAD\ ''x'',\\ \ LOAD\ ''y'',\\ \\ \qquad\qquad\ ,\\ \ LOAD\ ''z'',\\ \ LOAD\ ''a'',\\ \\ \\ ]\end{array}$$

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

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

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

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

$$\begin{array}{l} acomp\ a_1\ @ \\ acomp\ a_2\ @\ (\textsf{if}\ f\ \textsf{then}\ [JMPLESS\ n]\ \textsf{else}\ [JMPGE\ n]) \end{array}$$

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

$$\begin{array}{l} \textsf{let}\ cb_2 = bcomp\ b_2\ f\ n; \\ \quad m = \textsf{if}\ f\ \textsf{then}\ size\ cb_2\ \textsf{else}\ size\ cb_2 + n; \\ \quad cb_1 = bcomp\ b_1\ False\ m \\ \textsf{in}\ cb_1\ @\ cb_2 \end{array}$$

Correctness of $bcomp$

$$
\begin{array}{l}
0 \leq n \Longrightarrow \\
bcomp\ b\ f\ n \\
\vdash (0,\ s,\ stk) \rightarrow\!* \\
\quad (size\ (bcomp\ b\ f\ n) + (\textsf{\textit{if}}\ f = bval\ b\ s\ \textsf{\textit{then}}\ n\ \textsf{\textit{else}}\ 0), \\
\quad\ s,\ stk)
\end{array}
$$

Compiling com

$ccomp :: com \Rightarrow instr\ list$

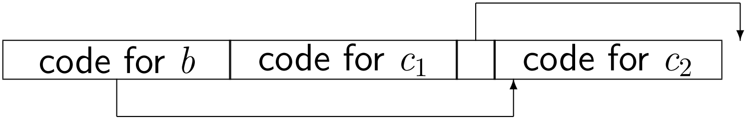
$ccomp\ SKIP = []$

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

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

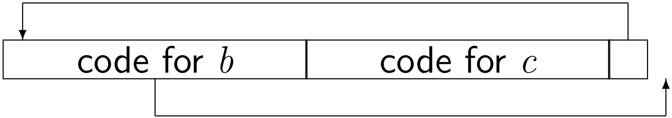
$$ccomp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$$

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



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

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



Correctness of $ccomp$

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

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

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: theory $Compiler2$

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

Chapter 9

Types

6 A Typed Version of IMP

7 Security Type Systems

6. A Typed Version of IMP

7. Security Type Systems

6 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 *exactly* the programs having a certain semantic property.

Automatic analysis of semantic program properties
is necessarily incomplete.

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

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

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

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

The other rules remain unchanged.

Example

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

- If $s\ ''x'' = \mathit{Iv}\ i$:
  $(c,\ s) \rightarrow (\mathit{SKIP},\ s(''x'' := \mathit{Iv}\ (i + 1)))$

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

6 A Typed Version of IMP

- Remarks on Type Systems
- Typed IMP: Semantics
- Typed IMP: Type System
- Type Safety of Typed 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 $''x''$ and $''y''$!

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

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

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 ($SKIP$, $::=$, $\dots$).
- Well-typedness of a term $C\ t_1 \dots t_n$ depends only on the well-typedness of its subterms $t_1, \dots, t_n$.

A syntax-directed set of rules

- is executable by backchaining without backtracking and
- backchaining terminates and requires at most as many steps as the size of the term.

Syntax-directedness

The big-step semantics is not syntax-directed:

- more than one rule per construct and
- the execution of $WHILE$ depends on the execution of $WHILE$.

6 A Typed Version of IMP

- Remarks on Type Systems
- Typed IMP: Semantics
- Typed IMP: Type System
- Type Safety of Typed IMP

Well-typed states

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

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

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:

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

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:

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

Progress:

$$\llbracket \Gamma \vdash b;\ \Gamma \vdash s \rrbracket \Longrightarrow \exists\, v.\ tbval\ b\ s\ v$$

$aexp$

Progress:

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

Preservation:

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

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.

6 A Typed Version of IMP

7 Security Type Systems

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.

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.

$sec :: aexp \Rightarrow level$

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

Security Levels on $bexp$

$sec :: bexp \Rightarrow level$

$$
\begin{aligned}
&sec\ (Bc\ v) = 0\\
&sec\ (Not\ b) = sec\ b\\
&sec\ (And\ b_1\ b_2) = max\ (sec\ b_1)\ (sec\ b_2)\\
&sec\ (Less\ a_1\ a_2) = max\ (sec\ a_1)\ (sec\ a_2)
\end{aligned}
$$

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 $c =$ $IF\ Less\ (V\ ''x1'')\ (V\ ''x'')$
$THEN\ ''x1'' ::= N\ 0$
$ELSE\ ''x1'' ::= N\ 1$

$1 \vdash c$? Yes

$2 \vdash c$? Yes

$3 \vdash c$? No

The type system

$$l \vdash SKIP$$

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

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

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

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

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.

7 Security Type Systems

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:

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

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

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

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

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

- 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 $\vdash$.

- 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: $0 \vdash' ''x'' ::= N\,1$ but not $\vdash ''x'' ::= N\,1 : 0$

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

Proof by induction.

7 Security Type Systems

Does noninterference really guarantee absence of information flow?

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

Beware of covert channels!

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

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.

Chapter 10

Data-Flow Analyses and Optimization

8 Definite Initialization Analysis

9 Live Variable Analysis

8 Definite Initialization Analysis

9 Live Variable 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 x < 1 THEN y := x ELSE y := x + 1;
y := y + 1
```

```
IF x < x THEN y := y + 1 ELSE y := x
```

Assume `x` and `y` are initialized:

```
WHILE x < y DO z := x; z := z + 1
```

Simplifying principle

We do not analyze boolean expressions to determine program execution.

8 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

8 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 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.

8 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

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

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

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

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

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

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

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

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

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

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:

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

$SKIP$ propagates:

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

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

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

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

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

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

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

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

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

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

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

We want in the end:

Well-initialized programs cannot go wrong.

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

We need to prove a generalized statement:

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

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

Proof needs some easy lemmas:

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

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

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

8 Definite Initialization Analysis

9 Live Variable Analysis

Motivation

Consider the following program:

```
x := y + 1;
y := y + 2;
x := y + 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 := 0`	☹
`y := x; y := 0; x := 0`	☺
`WHILE b DO y := x; x := 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

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

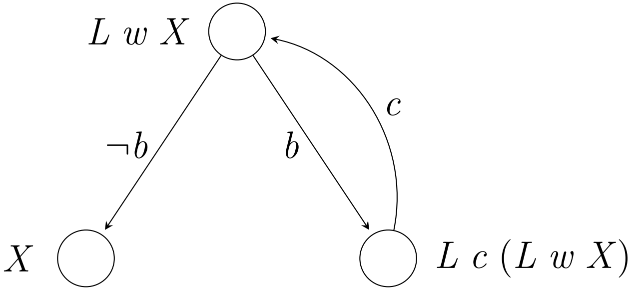
$$L\ c\ X\ =\ \text{live before } c \text{ relative to } X$$

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

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

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

$\Longrightarrow$

$vars\ b \subseteq L\ w\ X$ $\checkmark$

$X \subseteq L\ w\ X$ $\checkmark$

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

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

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

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

Example:

$$
\begin{array}{l}
L\ (\mathit{WHILE}\ \mathit{Less}\ (V\ ''x'')\ (V\ ''x'')\ \mathit{DO}\ ''y'' ::= V\ ''z'') \\
\quad \{''x''\}\ =\ \{''x'', ''z''\}
\end{array}
$$

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
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

$kill :: com \Rightarrow vname\ set$

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

$gen :: com \Rightarrow vname\ set$

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

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

$$L\ c\ X = gen\ c \cup (X - kill\ c)$$

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

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

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

9 Live Variable Analysis

Correctness 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 $Vars$):

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

Correctness 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 $L\ c\ X = gen\ c \cup (X - kill\ c)$:

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

9 Live Variable Analysis

- Correctness of L
- Dead Variable Elimination
- True Liveness
- Comparisons

Bury all assignments to dead variables:

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

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

$$
\begin{array}{l}
bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ = \\
\quad IF\ b\ THEN\ bury\ c_1\ X\ ELSE\ bury\ c_2\ X \\
bury\ (WHILE\ b\ DO\ c)\ X\ = \\
\quad WHILE\ b\ DO\ bury\ c\ (L\ (WHILE\ b\ DO\ c)\ X)
\end{array}
$$

Correctness of $bury$

$$bury\ c\ UNIV \sim c$$

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 correctness of L.

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

Follows from generalized statement:

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

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

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

Terminology

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

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

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

If $f\ x \leq x$ then x is a *pre-fixpoint* (*pfp*) of f.

If $x \leq y \Longrightarrow f\ x \leq f\ y$ for all x,y, then f is *monotone*.

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

Better:

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

But then

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

is not correct anymore.

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

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

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

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

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

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

$$\begin{aligned} b &= Less\ (N\ 0)\ (V\ y) \\ c &= y ::= V\ x;;\ x ::= V\ z \end{aligned}$$

$L\ w\ \{y\} = \{x,\ y\}$ 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 fixpoints

Theorem (Knaster-Tarski) Let $f :: \tau\ set \Rightarrow \tau\ set$.
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 pre-fixpoint and least fixpoint of f.

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\mathit{lfp}(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof
- $f\ (\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$
- $\mathit{lfp}\ f$ is the least pre-fixpoint of f

Lemma Let f be a monotone function on a partial order $\leq$. Then a least pre-fixpoint of f is also a least fixpoint.

Proof
- $f\ p \leq p \Longrightarrow f\ p = p$
- p is the least fixpoint

Definition of L

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

$L\ (WHILE\ b\ DO\ c)\ X = lfp\ f_w$
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 $f :: \tau\ set \Rightarrow \tau\ set$. 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 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 $w = \ WHILE\ b\ DO\ c$
where $b \ = \ Less\ (N\ 0)\ (V\ y)$
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`:

$\mathit{while} :: ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

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

Lemma Let $f :: \tau\ \mathit{set} \Rightarrow \tau\ \mathit{set}$. 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 $\mathit{lfp}\ f = \mathit{while}\ (\lambda X.\ f\ X \neq X)\ f\ \{\}$

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)

9 Live Variable Analysis

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.

Chapter 11

Denotational Semantics

10 A Relational Denotational Semantics of IMP

11 Continuity

What is it?

A denotational semantics maps syntax to semantics:

$$D :: syntax \Rightarrow meaning$$

Examples: $aval :: aexp \Rightarrow (state \Rightarrow val)$
$Big\_step :: com \Rightarrow (state \times state)\ set$

D must be defined by primitive recursion over the syntax

$$D\ (C\ t_1\ \dots\ t_n) = \dots\ (D\ t_1)\ \dots\ (D\ t_n)\ \dots$$

Fake: $Big\_step\ c = \{(s,t).\ (c,s) \Rightarrow t\}$

Why?

More abstract:

operational:	How to execute it
denotational:	What does it mean

Simpler proof principles:

operational:	relational, rule induction
denotational:	equational, structural induction

Relations

$$Id :: ('a \times 'a)\ set$$
$$Id = \{p.\ \exists\, x.\ p = (x,\ x)\}$$

$$(O) :: ('a \times 'b)\ set \Rightarrow ('b \times 'c)\ set \Rightarrow ('a \times 'c)\ set$$
$$r\ O\ s = \{(x,\ z).\ \exists\, y.\ (x,\ y) \in r \wedge (y,\ z) \in s\}$$

$$D :: com \Rightarrow com\_den$$

type_synonym $com\_den = (state \times state)\ set$

$$D\ SKIP = Id$$

$$D\ (x ::= a) = \{(s, t).\ t = s(x := aval\ a\ s)\}$$

$$D\ (c_1;;\ c_2) = D\ c_1\ O\ D\ c_2$$

$D\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$
$\{(s, t).$ *if* $bval\ b\ s$ *then* $(s, t) \in D\ c_1$ *else* $(s, t) \in D\ c_2\}$

Example

Let
$$\begin{aligned} c_1 &= ''x'' ::= N\ 0 \\ c_2 &= ''y'' ::= V\ ''x'': \end{aligned}$$

$$D\ c_1 = \{(s_1,s_2).\ s_2 = s_1(''x'' := 0)\}$$

$$D\ c_2 = \{(s_2,s_3).\ s_3 = s_2(''y'' := s_2\ ''x'')\}$$

$$D\ (c_1;;c_2) = \{(s_1,s_3).\ s_3 = s_1(''x'' := 0,\ ''y'' := 0)\}$$

$D\ (\mathit{WHILE}\ b\ \mathit{DO}\ c) = ?$

Wanted:

$D\ w =$
$\{(s,\ t).\ \textit{if}\ bval\ b\ s\ \textit{then}\ (s,\ t) \in D\ c\ O\ D\ w\ \textit{else}\ s = t\}$

Problem: not a denotational definition
not allowed by Isabelle

But $D\ w$ should be a solution of the equation.

General principle:

x is a solution of $x = f(x)$ $\longleftrightarrow$ x is a fixpoint of f

Define $D\ w$ as the least fixpoint of a suitable f

W

$D\ w =$
$\{(s,\ t).\ \textit{if}\ bval\ b\ s\ \textit{then}\ (s,\ t) \in D\ c\ O\ D\ w\ \textit{else}\ s = t\}$

$W ::$
$(state \Rightarrow bool) \Rightarrow com\_den \Rightarrow (com\_den \Rightarrow com\_den)$

$W\ db\ dc =$
$(\lambda dw.\ \{(s,\ t).\ \textit{if}\ db\ s\ \textit{then}\ (s,\ t) \in dc\ O\ dw\ \textit{else}\ s = t\})$

Lemma $W\ db\ dc$ is monotone.

We define

$$D\ (WHILE\ b\ DO\ c) = lfp\ (W\ (bval\ b)\ (D\ c))$$

By definition (where $f = W\ (bval\ b)\ (D\ c)$):

$$D\ w = lfp\ f = f\ (lfp\ f) = W\ (bval\ b)\ (D\ c)\ (D\ w)$$
$$= \{(s,\ t).\ \textit{if}\ bval\ b\ s\ \textit{then}\ (s,\ t) \in D\ c\ O\ D\ w\ \textit{else}\ s = t\}$$

Why least?

Formally: needed for equivalence proof with big-step.
An intuitive example:

$$w = \textit{WHILE Bc True DO SKIP}$$

Then

$$\begin{aligned} & W\ (bval\ (Bc\ True))\ (D\ SKIP) \\ =\ & W\ (\lambda s.\ True)\ Id \\ =\ & \lambda dw.\ \{(s,\ t).\ (s,\ t) \in Id\ O\ dw\} \\ =\ & \lambda dw.\ dw \end{aligned}$$

Every relation is a fixpoint!
Only the least relation $\{\}$ makes computational sense.

A denotational equivalence proof

Example

$$D\ w = D\ (IF\ b\ THEN\ c;;\ w\ ELSE\ SKIP)$$

where $w = WHILE\ b\ DO\ c$.

Let $f = W\ (bval\ b)\ (D\ c)$:

$D\ w$

$= \{(s,\ t).\ \textit{if}\ bval\ b\ s\ \textit{then}\ (s,\ t) \in D\ c\ O\ D\ w\ \textit{else}\ s = t\}$

$= D\ (IF\ b\ THEN\ c;;\ w\ ELSE\ SKIP)$

Equivalence of denotational and big-step semantics

Lemma $(c, s) \Rightarrow t \Longrightarrow (s, t) \in D\ c$
Proof by rule induction

Lemma $(s, t) \in D\ c \Longrightarrow (s, t) \in Big\_step\ c$
Proof by induction on c

Corollary $(s, t) \in D\ c \ \longleftrightarrow\ (c, s) \Rightarrow t$

Chains and continuity

Definition

$chain :: (nat \Rightarrow {'a}\ set) \Rightarrow bool$
$chain\ S = (\forall\, i.\ S\ i \subseteq S\ (Suc\ i))$

Definition (Continuous)

$cont :: ({'a}\ set \Rightarrow {'b}\ set) \Rightarrow bool$
$cont\ f = (\forall\, S.\ chain\ S \longrightarrow f\ (\bigcup_n S\ n) = (\bigcup_n f\ (S\ n)))$

Lemma $cont\ f \Longrightarrow mono\ f$

Kleene fixpoint theorem

Theorem $cont\ f \Longrightarrow lfp\ f = (\bigcup_n f^n\ \{\})$

Application to semantics

Lemma $W\ db\ dc$ is continuous.

Example

`WHILE x` $\neq$ `0 DO x := x - 1`

Semantics: $\{(s,t).\ 0 \leq s\ ''x'' \wedge t = s(''x'' := 0)\}$

Let $f = W\ db\ dc$
where $db = bval\ b = (\lambda s.\ s\ ''x'' \neq 0)$
$\qquad dc = D\ c = \quad \{(s,\ t).\ t = s(''x'' := s\ ''x'' - 1)\}$

A proof of determinism

$single\_valued\ r =$
$(\forall\, x\ y\ z.\ (x,\ y) \in r \wedge (x,\ z) \in r \longrightarrow y = z)$

Lemma If $f :: com\_den \Rightarrow com\_den$ is continuous and preserves single-valuedness then $lfp\ f$ is single-valued.

Lemma $single\_valued\ (D\ c)$

Chapter 12

Hoare Logic

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

12 Partial Correctness

- Introduction
- The Syntactic Approach
- The Semantic Approach
- Soundness and Completeness

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:

$''y'' ::= N\ 0;;\ wsum$

where

$$
\begin{aligned}
&wsum \equiv\\
&WHILE\ Less\ (N\ 0)\ (V\ ''x'')\\
&DO\ (''y'' ::= Plus\ (V\ ''y'')\ (V\ ''x'');;\\
&\qquad ''x'' ::= Plus\ (V\ ''x'')\ (N\ (-\ 1)))
\end{aligned}
$$

At the end of the execution of $''y'' ::= N\ 0;;\ wsum$ variable $''y''$ should contain the sum $1 + \ldots + i$ where i is the initial value of $''x''$.

$sum\ i = (\textit{if}\ i \le 0\ \textit{then}\ 0\ \textit{else}\ sum\ (i - 1) + i)$

A proof via operational semantics

Theorem:

$$(''y'' ::= N\,0;;\ wsum,\ s) \Rightarrow t \Longrightarrow$$
$$t\ ''y'' = sum\ (s\ ''x'')$$

Required Lemma:

$$(wsum,\ s) \Rightarrow t \Longrightarrow$$
$$t\ ''y'' = s\ ''y'' + sum\ (s\ ''x'')$$

Proved by rule induction.

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

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

But no free lunch:

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

This is the standard approach.

Formulas are syntactic objects.

Everything is very concrete and simple.

But complex to formalize.

Hence we soon move to a semantic view of formulas.

Reason for introduction of syntactic approach: didactic

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

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

- P and Q are *syntactic formulas* involving program variables
- P is the *precondition*, Q is the *postcondition*
- $\{P\}\ c\ \{Q\}$ means that if P is true at the start of the execution, then 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

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

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

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

Boundary cases:

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

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

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

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:

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

Alternative explanation of assignment rule:

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

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.

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

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

Example

$$\{x = i\}$$

$$y := 0;$$
$$WHILE\ 0 < x\ DO\ (y := y{+}x;\ x := x{-}1)$$

$$\{y = sum\ i\}$$

$$\mathit{wsum} = \mathit{WHILE}\ x > 0\ \mathit{DO}\ (y := y + x; x := x - 1)$$

$$\dfrac{\dfrac{x = i \xrightarrow{\checkmark} I[0/y]}{\{x = i\}y := 0\{I\}} \qquad \dfrac{\dfrac{\dfrac{I \wedge x > 0 \xrightarrow{\checkmark} I[x-1/x][y+x/y]}{\{I \wedge x > 0\}y := y + x\{I[x-1/x]\}x := x - 1\{I\}}}{\{I \wedge x > 0\}y := y + x; x := x - 1\{I\}} \quad I \wedge x \leq 0 \xrightarrow{\checkmark} y = \mathit{sum}\, i}{\{I\}\mathit{wsum}\{y = \mathit{sum}\, i\}}}{\{x = i\}y := 0; \mathit{wsum}\{y = \mathit{sum}\, i\}}$$

$$I \quad = \quad y = \mathit{sum}\, i - \mathit{sum}\, x$$

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.

12 Partial Correctness

Assertions are predicates on states

$$assn = state \Rightarrow bool$$

Alternative view: *sets of states*

Semantic approach simplifies meta-theory, our main objective.

Validity

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

$$\longleftrightarrow$$

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

"$\{P\}\ c\ \{Q\}$ 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 $\begin{aligned} P &= (\lambda s.\ (\lambda t.\ t\ x = 5)(s[Plus\ (V\ x)\ (N\ 5)/x])) \\ &= (\lambda s.\ (\lambda t.\ t\ x = 5)(s(x := s\ x + 5))) \\ &= (\lambda s.\ s\ x + 5 = 5) \end{aligned}$

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

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

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

$$\frac{\begin{array}{c}\forall\, s.\ P'\ s \longrightarrow P\ s \\ \vdash \{P\}\ c\ \{Q\} \\ \forall\, s.\ Q\ s \longrightarrow Q'\ s\end{array}}{\vdash \{P'\}\ c\ \{Q'\}}$$

Hoare_Examples.thy

12 Partial Correctness

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

$wp\ SKIP\ Q = Q$

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

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

$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q =$
$(\lambda s.\ \textsf{if}\ bval\ b\ s\ \textsf{then}\ wp\ c_1\ Q\ s\ \textsf{else}\ wp\ c_2\ Q\ s)$

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

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

Completeness

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

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

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

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

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

Completeness

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

$$\dfrac{\dfrac{\dfrac{\forall s.\ \begin{array}{l} wp\ w\ Q\ s \wedge bval\ b\ s \\ \longrightarrow wp\ c\ (wp\ w\ Q)\ s \end{array} \qquad \dfrac{IH}{\vdash \{wp\ c\ (wp\ w\ Q)\}\ c\ \{wp\ w\ Q\}}}{\vdash \{\lambda s.\ wp\ w\ Q\ s \wedge bval\ b\ s\}\ c\ \{wp\ w\ Q\}}}{\vdash \{wp\ w\ Q\}\ w\ \{\lambda t.\ wp\ w\ Q\ t \wedge \neg bval\ b\ t\}}}{\vdash \{wp\ w\ Q\}\ w\ \{Q\}}$$

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

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:

From $\{P\}\ c\ \{Q\}$ *generate an assertion* A*, the verification condition, such that* $\vdash \{P\}\ c\ \{Q\}$ *iff* A *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:

$$\text{VCG} = \text{Verification Condition Generator}$$

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: $vc :: com \Rightarrow assn \Rightarrow bool$

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 $While$:

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

Concrete syntax: like commands, except for $WHILE$:

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

Weakest precondition

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

$pre\ SKIP\ Q = Q$

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

$pre\ (C_1;;\ C_2)\ Q = pre\ C_1\ (pre\ C_2\ Q)$

$pre\ (IF\ b\ THEN\ C_1\ ELSE\ C_2)\ Q =$
$(\lambda s.\ \textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ pre\ C_1\ Q\ s\ \textsf{\textit{else}}\ pre\ C_2\ Q\ s)$

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

Warning

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

Verification condition

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

$vc\ SKIP\ Q = True$

$vc\ (x ::= a)\ Q = True$

$vc\ (C_1;;\ C_2)\ Q = (vc\ C_1\ (pre\ C_2\ Q) \wedge vc\ C_2\ Q)$

$vc\ (IF\ b\ THEN\ C_1\ ELSE\ C_2)\ Q =$
$(vc\ C_1\ Q \wedge vc\ C_2\ Q)$

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

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

$$strip :: acom \Rightarrow com$$

$$strip\ SKIP = SKIP$$
$$strip\ (x ::= a) = x ::= a$$
$$strip\ (C_1;;\ C_2) = strip\ C_1;;\ strip\ C_2$$
$$strip\ (IF\ b\ THEN\ C_1\ ELSE\ C_2) =$$
$$IF\ b\ THEN\ strip\ C_1\ ELSE\ strip\ C_2$$
$$strip\ (\{I\}\ WHILE\ b\ DO\ C) = WHILE\ b\ DO\ strip\ C$$

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

$$vc\ C\ Q \Longrightarrow \vdash \{pre\ C\ Q\}\ strip\ C\ \{Q\}$$

Proof by induction on C, for arbitrary Q.

Corollary:

$$\llbracket vc\ C\ Q;\ \forall\, s.\ P\ s \longrightarrow pre\ C\ Q\ s \rrbracket$$
$$\Longrightarrow \vdash \{P\}\ strip\ C\ \{Q\}$$

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

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

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

$$\llbracket vc\ C\ Q;\ \forall s.\ P\ s \longrightarrow pre\ C\ Q\ s \rrbracket$$
$$\Longrightarrow \vdash \{P\}\ strip\ C\ \{Q\}$$

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. $\vdash$

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

Proof by rule induction.
Needs two monotonicity lemmas:

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

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

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

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

Total Correctness = Partial Correctness + Termination

Formally:

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

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

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

$WHILE$ rule can be generalized from a function to a relation:

$$\frac{\bigwedge n.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge\ T\ s\ n\}\ c\ \{\lambda s.\ P\ s \wedge (\exists\, n'{<}n.\ T\ s\ n')\}}{\vdash_t \{\lambda s.\ P\ s \wedge (\exists\, n.\ T\ s\ n)\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}}$$

Hoare_Total.thy

Example

Soundness

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

Proof by induction, with a nested induction on n 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

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

Proof of $\vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$ is by induction on c.
In the $WHILE\ b\ DO\ c$ case, use the $WHILE$ rule with

$$\frac{\neg\ bval\ b\ s}{T\ s\ 0} \qquad \frac{bval\ b\ s \qquad (c,\ s) \Rightarrow s' \qquad T\ s'\ n}{T\ s\ (n+1)}$$

$T\ s\ n$ means that $WHILE\ b\ DO\ c$ started in state s needs n iterations to terminate.

Chapter 13

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]` $\leadsto$
  `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 $0 \leq m \wedge n <$ `a.length`
  - ...
- Null pointer analysis
- ...

Precision

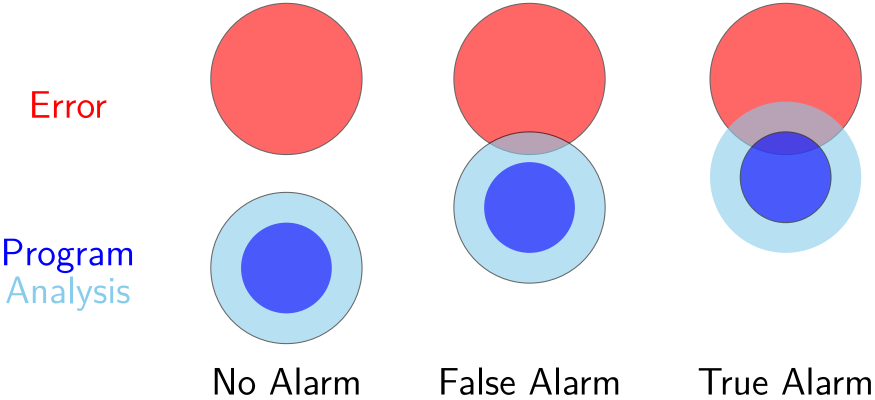
A consequence of Rice's theorem:

> *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

- 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

$$\{ \ \ldots \ \}$$

program text with semantic information.

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

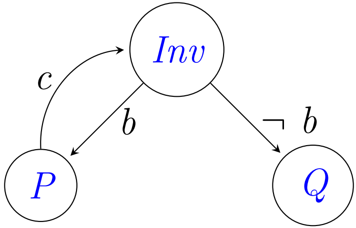
$$\texttt{x := 0} \ \{ \ \ldots \ \}; \ \texttt{y := 0} \ \{ \ \ldots \ \}$$

Annotated WHILE

View

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

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

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

$'a$: type of annotations

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

Abstract syntax

datatype

$$
\begin{array}{ll}
\multicolumn{2}{l}{'a\ acom =} \\
& SKIP\ \ 'a \\
| & Assign\ \ string\ \ aexp\ \ 'a \\
| & Seq\ \ ('a\ acom)\ \ ('a\ acom) \\
| & If\ \ bexp\ \ 'a\ \ ('a\ acom)\ \ 'a\ \ ('a\ acom)\ \ 'a \\
| & While\ \ 'a\ \ bexp\ \ 'a\ \ ('a\ acom)\ \ 'a
\end{array}
$$

Auxiliary functions: $strip$

Strips all annotations from an annotated command

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

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

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

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

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

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

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

Auxiliary functions: $annos$

The list of annotations in an annotated command (from left to right)

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

$$annos\ (SKIP\ \{P\}) = [P]$$
$$annos\ (x ::= e\ \{P\}) = [P]$$
$$annos\ (C_1;;\ C_2) = annos\ C_1\ @\ annos\ C_2$$
$$annos\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$$
$$P_1\ \#\ annos\ C_1\ @\ P_2\ \#\ annos\ C_2\ @\ [Q]$$
$$annos\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) =$$
$$I\ \#\ P\ \#\ annos\ C\ @\ [Q]$$

Auxiliary functions: $anno$

$anno :: {'a}\ acom \Rightarrow nat \Rightarrow {'a}$
$anno\ C\ p = annos\ C\ !\ p$

The p-th annotation (starting from 0)

Auxiliary functions: $post$

$post :: {'a}\ acom \Rightarrow {'a}$
$post\ C = last\ (annos\ C)$

The rightmost/last/post annotation

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

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.

$\underset{\text{flowing into the command}}{\overset{\text{start states}}{\vee}}$

$step$

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

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

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

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

$step$

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

$\{S \cup post\ C\}$
$WHILE\ b$
$DO\ \{\{s \in I.\ bval\ b\ s\}\}$
$\quad step\ P\ C$
$\{\{s \in I.\ \neg\ bval\ b\ s\}\}$

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 fixpoint:

$$step\ S\ C = C$$

Collecting_Examples.thy

Abstract example

Let $C =$ `{ ` I ` }`

```
{ I }
WHILE b
DO { P } C0
{ Q }
```

$step\ S\ C = C$ means

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

Fixpoint = solution of equation system

Iteration is just one way of solving equations

Why *least* fixpoint?

```
{ I }
WHILE true
DO { I } SKIP { I }
{ {} }
```

Is fixpoint of $step\ \{\}$ for every I

But the "reachable" fixpoint is $I = \{\}$

Does $step$ always have a least fixpoint?

Partial order

A type $'a$ is a *partial order* if

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

Complete lattice

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

- $\forall\, s{\in}S.\ l \le s$
- If $\forall\, s{\in}S.\ l' \le s$ then $l' \le l$

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

Fact Type $'a\ set$ is a complete lattice
where $\le\ =\ \subseteq$ and $\bigsqcap\ =\ \bigcap$

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 fixpoints

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 (pre-)fixpoint

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

Proof just like the version for sets.

Ordering $'a\ acom$

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

$$C_1 \le C_2 \longleftrightarrow$$
$$strip\ C_1 = strip\ C_2 \wedge$$
$$(\forall\, p{<}length\ (annos\ C_1).\ anno\ C_1\ p \le anno\ C_2\ p)$$

Lemma If $'a$ is a partial order, so is $'a\ acom$.

Ordering $'a\ acom$

Example:

$$x ::= N\ 0\ \{\{a\}\} \le x ::= N\ 0\ \{\{a,\ b\}\} \quad \longleftrightarrow \quad \mathit{True}$$
$$x ::= N\ 0\ \{\{a\}\} \le x ::= N\ 0\ \{\{\}\} \quad \longleftrightarrow \quad \mathit{False}$$
$$x ::= N\ 0\ \{S\} \le x ::= N\ 1\ \{S\} \quad \longleftrightarrow \quad \mathit{False}$$

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,
$f \in A \to B$ means $\forall\, a{\in}A.\ f\ a \in B$.

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

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

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 . . .

Auxiliary function: $annotate$

$annotate :: (nat \Rightarrow {'a}) \Rightarrow com \Rightarrow {'a}\ acom$

Set annotation number p (as counted by $anno$) to $f\ p$.
Definition is technical. The characteristic lemma:

$anno\ (annotate\ f\ c)\ p = f\ p$

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

$$annotate\ (\lambda p. \bigsqcap \{anno\ C\ p\ | C.\ C \in M\})\ c$$

Proof straightforward (pointwise).

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

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

Proof of

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

uses

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

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:

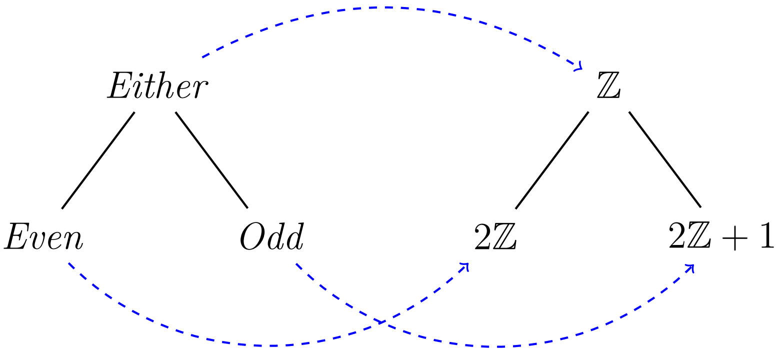
$$val\ set \quad \rightsquigarrow \quad 'av$$

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

Example: parity

$$
\begin{tikzcd}
& Either & \\
Even \arrow[ur, no head, "\leq"] & & Odd \arrow[ul, no head, "\leq"']
\end{tikzcd}
$$

Fact Type $parity$ is a partial order.

Top element

A partial order $'a$ has a *top element* $\top :: 'a$ if

$$a \leq \top$$

Semilattice

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

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

  $$x \leq x \sqcup y \qquad y \leq x \sqcup y$$

  $$[\![x \leq z;\ y \leq z]\!] \Longrightarrow x \sqcup y \leq z$$

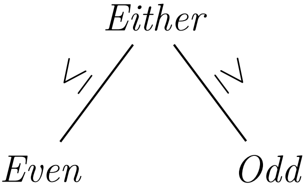
Application: abstract $\cup$, join two computation paths
We often call $\sqcup$ the *join* operation.

$\leq$ uniquely determines $\sqcup$

Fact If $'a$ is a semilattice, then the least upper bound of two elements is uniquely determined.

If u_1 and u_2 are least upper bounds of x and y,
then $u_1 \leq u_2$ and $u_2 \leq u_1$.

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 $order$:	partial orders
class $semilattice\_sup$:	semilattices
class $semilattice\_sup\_top$:	semilattices with top element

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: $parity :: semilattice\_sup$

HOL/Orderings.thy
Abs_Int1_parity.thy

Orderings and instances

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 $'a :: semilattice\_sup\_top$
then $'b \Rightarrow 'a :: semilattice\_sup\_top$

Proof

$(f \leq g) = (\forall x.\ f\ x \leq g\ x)$
$f \sqcup g = (\lambda x.\ f\ x \sqcup g\ x)$
$\top = (\lambda x.\ \top)$

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 $'a :: semilattice\_sup\_top$
then $'a\ option :: semilattice\_sup\_top$

Proof

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

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

$\top = Some\ \top$

Corollary If $'a :: semilattice\_sup\_top$
then $'a\ st\ option :: semilattice\_sup\_top$

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

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

$'a\ acom$

Remember:

Lemma If $'a :: order$ then $'a\ acom :: order$.

Partial order 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 $'av :: semilattice\_sup\_top$
Concretization function: $\gamma :: 'av \Rightarrow val\ set$
Assumptions: $a \leq b \Longrightarrow \gamma\ a \subseteq \gamma\ b$
$\gamma\ \top = UNIV$

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

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

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

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

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 $parity$

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

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

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

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

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

Idea: define both as instances of a generic step function:

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

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

Parameterized wrt

- type $'a$ with $\sqcup$
- the interpretation of assignments and tests:
  $asem :: vname \Rightarrow aexp \Rightarrow {}'a \Rightarrow {}'a$
  $bsem :: bexp \Rightarrow {}'a \Rightarrow {}'a$

$$Step\ a\ (SKIP\ \{\_\}) = SKIP\ \{a\}$$

$$Step\ a\ (x ::= e\ \{\_\}) = x ::= e\ \{asem\ x\ e\ a\}$$

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

$$Step\ a\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{\_\}) =$$
$$IF\ b\ THEN\ \{bsem\ b\ a\}\ Step\ P_1\ C_1$$
$$ELSE\ \{bsem\ (Not\ b)\ a\}\ Step\ P_2\ C_2$$
$$\{post\ C_1 \sqcup post\ C_2\}$$

$$Step\ a\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{\_\}) =$$
$$\{a \sqcup post\ C\}\ WHILE\ b\ DO\ \{bsem\ b\ I\}\ Step\ P\ C$$
$$\{bsem\ (Not\ b)\ I\}$$

Instantiating *Step*

The truth: $asem$ and $bsem$ are (hidden) parameters of $Step$: $Step\ asem\ bsem\ \ldots$

$step =$
$Step\ (\lambda x\ e\ S.\ \{s(x := aval\ e\ s)\ |s.\ s \in S\})$
$(\lambda b\ S.\ \{s \in S.\ bval\ b\ s\})$

$step' = Step\ asem\ (\lambda b\ S.\ S)$
where
$asem\ x\ e\ S =$
$(\textsf{case}\ S\ \textsf{of}\ None \Rightarrow None$
$|\ Some\ S \Rightarrow Some\ (S(x := aval'\ e\ S)))$

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 $step$:

Corollary $step\ (\gamma_o\ S)\ (\gamma_c\ C) \le \gamma_c\ (step'\ S\ C)$

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

Lemma $Step\ f\ g\ (\gamma_o\ S)\ (\gamma_c\ C) \le \gamma_c\ (Step\ f'\ g'\ S\ C)$
if for all x, e, b: $f\ x\ e\ (\gamma_o\ S) \subseteq \gamma_o\ (f'\ x\ e\ S)$
$g\ b\ (\gamma_o\ S) \subseteq \gamma_o\ (g'\ b\ S)$

Proof by an easy induction on C

The abstract interpreter

- Ideally: iterate $step'$ until a fixpoint is reached
- May take too long
- Sufficient: any pre-fixpoint: $step'\ S\ C \leq C$
  Means iteration does not increase annotations,
  i.e. annotations are consistent but maybe too big

Unbounded search

From the HOL library:

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

such that

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

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

Pre-fixpoint:

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

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

Start iteration with least annotated command:

$bot\ c = annotate\ (\lambda p.\ None)\ c$

The generic abstract interpreter

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

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

Proof From the assumption: $step'\ \top\ C \leq C$
By monotonicity: $\gamma_c\ (step'\ \top\ C) \leq \gamma_c\ C$
By $step/step'$: $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ \top\ C)$
Hence $\gamma_c\ C$ is a pfp of $step\ (\gamma_o\ \top) = step\ UNIV$
Because CS is the least pfp of $step\ UNIV$: $CS\ c \leq \gamma_c\ C$

Problem

AI is not directly executable

because pfp compares $f\ C \le C$
where $C :: 'av\ st\ option\ acom$
which compares functions $vname \Rightarrow 'av$
which is not computable: $vname$ is infinite.

Solution

Record only the *finite* set of variables actually present in a program.

An association list representation:

type_synonym $'a\ st\_rep = (vname \times\ 'a)\ list$

From $'a\ st\_rep$ back to $vname \Rightarrow\ 'a$:

fun $fun\_rep :: ('a{::}top)\ st\_rep \Rightarrow (vname \Rightarrow\ 'a)$
$fun\_rep\ ((x,\ a)\ \#\ ps) = (fun\_rep\ ps)(x := a)$
$fun\_rep\ [] = (\lambda x.\ \top)$

Missing variables are mapped to $\top$

Example: $fun\_rep\ [(''x'',\ a),\ (''x'',\ b)]$
$= ((\lambda x.\ \top)(''x'' := b))(''x'' := a) = (\lambda x.\ \top)(''x'' := a)$

Comparing association lists

Compare them only on their finite "domains":

$$less\_eq\_st\_rep\ ps_1\ ps_2 =$$
$$(\forall\, x{\in}set\ (map\ fst\ ps_1) \cup set\ (map\ fst\ ps_2).$$
$$\quad fun\_rep\ ps_1\ x \le fun\_rep\ ps_2\ x)$$

Not a partial order because not antisymmetric!

Example: $[(''x'',\ a),\ (''y'',\ b)]$ and $[(''y'',\ b),\ (''x'',\ a)]$

Quotient type $'a\ st$

Define $eq\_st\ ps_1\ ps_2 = (fun\_rep\ ps_1 = fun\_rep\ ps_2)$

Overwrite $'a\ st = vname \Rightarrow 'a$ by

quotient_type $'a\ st = ('a{::}top)\ st\_rep\ /\ eq\_st$

Elements of $'a\ st$:

equivalence classes $[ps]_{\mathit{eq\_st}} = \{ps'.\ eq\_st\ ps\ ps'\}$

Abbreviate $[ps]_{\mathit{eq\_st}}$ by $St\ ps$

Alternative to quotient: canonical representatives

For example, the subtype of sorted association lists:

- $[(''x'',\ a),\ (''y'',\ b)]$
- $[(''y'',\ b),\ (''x'',\ a)]$

More concrete, and probably a bit more complicated

Auxiliary functions on $'a\ st$

Turning an abstract state into a function:

$fun\ (St\ ps) = fun\_rep\ ps$

Updating an abstract state:

$update\ (St\ ps)\ x\ a = St\ ((x,\ a)\ \#\ ps)$

Turning $'a\ st$ into a semilattice

$$(St\ ps_1 \leq St\ ps_2) = less\_eq\_st\_rep\ ps_1\ ps_2$$

$$St\ ps_1 \sqcup St\ ps_2 = St(map2\_st\_rep\ (\sqcup)\ ps_1\ ps_2)$$

fun $map2\_st\_rep ::$
$('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ st\_rep \Rightarrow 'a\ st\_rep \Rightarrow 'a\ st\_rep$

Characteristic property:

$$fun\_rep\ (map2\_st\_rep\ f\ ps_1\ ps_2) =$$
$$(\lambda x.\ f\ (fun\_rep\ ps_1\ x)\ (fun\_rep\ ps_2\ x))$$

if $f\ \top\ \top = \top$

Modified abstract interpreter

Everything as before, except for $S :: {'av}\ st$:

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

Now AI is executable!

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* pre-fixpoints

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

Monotonicity

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

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

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

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

and therefore $step'$ is also monotone:

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

Precision: smaller is better

If f is monotone and $\perp$ is a least element,
then $pfp\ f \perp$ is a least pre-fixpoint of f

Lemma Let $\le$ be a partial order on a set L with least element $\bot \in L$: $x \in L \Longrightarrow \bot \le x$.
Let $f \in L \to L$ be a monotone function.
If $while\_option\ (\lambda x.\ \neg\ f\ x \le x)\ f\ \bot = Some\ p$ then p is the least pre-fixpoint of f on L.
That is, if $f\ q \le q$ for some $q \in L$, then $p \le q$.

Proof Clearly $f\ p \le p$.
Given any pre-fixpoint $q \in L$, property

$$P\ x = (x \in L \wedge x \le q)$$

is an invariant of the while loop:

$P\ \bot$ holds and $P\ x$ implies $f\ x \le f\ q \le q$

Hence upon termination $P\ p$ must hold and thus $p \le q$.

Application to

$$AI\ c = pfp\ (step'\ \top)\ (bot\ c)$$
$$pfp\ f = while\_option\ (\lambda x.\ \neg\ f\ x \leq x)\ f$$

Because $bot\ c$ is a least element and $step'$ is monotone, AI returns least pre-fixpoints

Termination

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

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

Modulo some details ...

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: $m\ x \leq h$

$x < y \Longrightarrow m\ y < m\ x$

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

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

Measure functions

$$m_c :: {}'av\ st\ option\ acom \Rightarrow nat$$
$$m_c\ C = (\sum a{\leftarrow}annos\ C.\ m_o\ a\ (vars\ C))$$

$$m_o :: {}'av\ st\ option \Rightarrow vname\ set \Rightarrow nat$$
$$m_o\ (Some\ S)\ X = m_s\ S\ X$$
$$m_o\ None\ X = h * card\ X + 1$$

$$m_s :: {}'av\ st \Rightarrow vname\ set \Rightarrow nat$$
$$m_s\ S\ X = (\sum x{\in}X.\ m\ (S\ x))$$

All measure functions are bounded:

$$finite\ X \Longrightarrow m_s\ S\ X \leq h * card\ X$$
$$finite\ X \Longrightarrow m_o\ opt\ X \leq h * card\ X + 1$$
$$m_c\ C \leq length\ (annos\ C) * (h * card\ (vars\ C) + 1)$$

Hence $AI\ c$ requires at most $p * ((h + 1) * n + 1)$ steps where $p =$ the number of annotation points of c and $n =$ the number of variables in c.

Complication

Anti-monotonicity does not hold!

Example:

$\mathit{finite}\ X \Longrightarrow S_1 < S_2 \Longrightarrow m_s\ S_2\ X < m_s\ S_1\ X$

because $S_1 < S_2 \longleftrightarrow S_1 \leq S_2 \wedge (\exists\, x.\ S_1\ x < S_2\ x)$

Need to know that S_1 and S_2 are the same outside X.
Follows if both are $\top$ outside X.

top_on

$$top\_on_s :: 'av\ st \Rightarrow\ vname\ set \Rightarrow\ bool$$
$$top\_on_s\ S\ X = (\forall\, x{\in}X.\ S\ x = \top)$$

$$top\_on_o :: 'av\ st\ option \Rightarrow\ vname\ set \Rightarrow\ bool$$
$$top\_on_o\ (Some\ S)\ X = top\_on_s\ S\ X$$
$$top\_on_o\ None\ X = True$$

$$top\_on_c :: 'av\ st\ option\ acom \Rightarrow\ bool$$
$$top\_on_c\ C\ X = (\forall\, a{\in}set\ (annos\ C).\ top\_on_o\ a\ X)$$

Now we can formulate and prove anti-monotonicity:

$$\llbracket \mathit{finite}\ X;\ S_1 = S_2\ \mathit{on} -\ X;\ S_1 < S_2 \rrbracket$$
$$\Longrightarrow m_s\ S_2\ X < m_s\ S_1\ X$$

$$\llbracket \mathit{finite}\ X;\ \mathit{top\_on}_o\ o_1\ (-\ X);\ \mathit{top\_on}_o\ o_2\ (-\ X);$$
$$o_1 < o_2 \rrbracket$$
$$\Longrightarrow m_o\ o_2\ X < m_o\ o_1\ X$$

$$\llbracket \mathit{top\_on}_c\ C_1\ (-\ \mathit{vars}\ C_1);\ \mathit{top\_on}_c\ C_2\ (-\ \mathit{vars}\ C_2);$$
$$C_1 < C_2 \rrbracket$$
$$\Longrightarrow m_c\ C_2 < m_c\ C_1$$

Now we can prove termination

$$\exists\, C.\ AI\ c = Some\ C$$

because $step'$ leaves top_on_s invariant:

$$top\_on_c\ C\ (-\ vars\ C) \Longrightarrow$$
$$top\_on_c\ (step'\ \top\ C)\ (-\ vars\ C)$$

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 $S' \leq S$ such that

if $s \in \gamma_s\ S$ and $bval\ b\ s$ 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* if

- it is a semilattice
- there is a greatest lower bound operation $\sqcap :: {'a} \Rightarrow {'a} \Rightarrow {'a}$

  $$x \sqcap y \leq x \qquad x \sqcap y \leq y$$

  $$[\![z \leq x;\ z \leq y]\!] \Longrightarrow z \leq x \sqcap y$$

Note: $\sqcap$ is also called *infimum* or *meet*.

Type class: $lattice$

Bounded lattice

A type $'a$ is a *bounded lattice* if

- it is a lattice
- there is a top element $\top :: 'a$
- and a *bottom* element $\bot :: 'a$
  $\bot \leq a$

Type class: $bounded\_lattice$

Fact Any complete lattice is a bounded lattice.

Concretization

We strengthen the abstract interpretation framework by assuming

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

  $\Longrightarrow \gamma\ (a_1 \sqcap a_2) = \gamma\ a_1 \cap \gamma\ a_2$
  $\Longrightarrow \sqcap$ 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 $S' \leq S$ such that

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

$\gamma_s\ S'$ overapproximates the subset of $\gamma_s\ S$ that makes e evaluate to an $\in \gamma\ a$.

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

$inv\_aval'\ N$

$inv\_aval' ::$

$$aexp \Rightarrow {'av} \Rightarrow {'av}\ st\ option \Rightarrow {'av}\ st\ option$$

$inv\_aval'\ (N\ n)\ \ a\ S =$
$(\textit{if}\ test\_num'\ n\ a\ \textit{then}\ S\ \textit{else}\ None)$

An extension of the interface of our framework:

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

Assumption:

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

Note: $i \in \gamma\ \ a$ not necessarily executable

inv_aval' V

inv_aval' $(V\ x)$ a $S =$

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

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

$inv\_aval'\ Plus$

A further extension of the interface of our framework:

$inv\_plus' :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$

Assumption:

$$
\begin{array}{l}
inv\_plus'\ a\ a_1\ a_2 = (a_1{}',\ a_2{}') \Longrightarrow \\
\quad \gamma\ a_1{}' \supseteq \{i_1 \in \gamma\ a_1.\ \exists\, i_2 \in \gamma\ a_2.\ i_1{+}i_2 \in \gamma\ a\} \wedge \\
\quad \gamma\ a_2{}' \supseteq \{i_2 \in \gamma\ a_2.\ \exists\, i_1 \in \gamma\ a_1.\ i_1{+}i_2 \in \gamma\ a\}
\end{array}
$$

Definition:

$$
\begin{array}{l}
inv\_aval'\ (Plus\ e_1\ e_2)\ a\ S = \\
(\textsf{\textit{let}}\ (a_1,\ a_2) = inv\_plus'\ a\ (aval''\ e_1\ S)\ (aval''\ e_2\ S) \\
\ \textsf{\textit{in}}\ inv\_aval'\ e_1\ a_1\ (inv\_aval'\ e_2\ a_2\ S))
\end{array}
$$

(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 $S' \leq S$ such that

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

$\gamma_s\ S'$ overapproximates the subset of $\gamma_s\ S$
that makes b evaluate to res.

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

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

$$inv\_bval'\ (Not\ b)\ res\ S = inv\_bval'\ b\ (\neg\ res)\ S$$

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

$$inv\_bval'\ (Less\ e_1\ e_2)\ res\ S =$$
$$\textsf{let}\ (a_1,\ a_2) = inv\_less'\ res\ (aval''\ e_1\ S)\ (aval''\ e_2\ S)$$
$$\textsf{in}\ inv\_aval'\ e_1\ a_1\ (inv\_aval'\ e_2\ a_2\ S)$$

A further extension of the interface of our framework:

$inv\_less' :: bool \Rightarrow {}'av \Rightarrow {}'av \Rightarrow {}'av \times {}'av$

Assumption:

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

Example: intervals, informally

$$inv\_plus'\ [0,4]\ [10,20]\ [-10,0] = ([10,14],[-10,-6])$$

$$inv\_less'\ True\ [0,20] \quad [-5,5] = ([0,4],[1,5])$$

$$inv\_bval'\ (\texttt{x + y < z})\ True$$
$$\{\texttt{x} \mapsto [10,20], \texttt{y} \mapsto [-10,0], \texttt{z} \mapsto [-5,5]\}:$$

$$inv\_aval'\ \texttt{z}\ [1,5]\ \{\bullet\} = \{\bullet, \texttt{z} \mapsto [1,5]\}$$

$$inv\_aval'\ (\texttt{x + y})\ [0,4]\ \{\bullet\}:$$

$$inv\_aval'\ \texttt{y}\ [-10,-6]\ \{\bullet\} = \{\bullet, \texttt{y} \mapsto [-10,-6], \bullet\}$$

$$inv\_aval'\ \texttt{x}\ [10,14]\ \{\bullet\} =$$
$$\{\texttt{x} \mapsto [10,14], \texttt{y} \mapsto [-10,-6], \texttt{z} \mapsto [1,5]\}$$

$step'$

Before: $step' = Step\ asem\ (\lambda b\ S.\ S)$
Now: $step' = Step\ asem\ (\lambda b.\ inv\_bval'\ b\ True)$

Correctness proof

Almost as before, but with correctness lemmas for inv_aval'

$\{s \in \gamma_o\ S.\ aval\ e\ s \in \gamma\ a\} \subseteq \gamma_o\ (inv\_aval'\ e\ a\ S)$

and inv_bval':

$\{s \in \gamma_o\ S.\ bv = bval\ b\ s\} \subseteq \gamma_o\ (inv\_bval'\ b\ bv\ S)$

Summary

Extended interface to abstract interpreter:

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

∞ and $-\infty$

Extending type $'a$ with ∞ and $-\infty$:

datatype $'a\ extended = Fin\ 'a \mid \infty \mid -\infty$

type_synonym $eint = int\ extended$

$(+)$, $(-)$, $(\leq)$, $(<)$ extended to $eint$

Intervals

datatype $'a\ extended = Fin\ 'a \mid \infty \mid -\infty$
type_synonym $eint = int\ extended$

A simple model of intervals:
type_synonym $eint2 = eint \times eint$

$\gamma\_rep :: eint2 \Rightarrow int\ set$
$\gamma\_rep\ (l, h) = \{i.\ l \leq Fin\ i \wedge Fin\ i \leq h\}$

Problem:
infinitely many empty intervals: all (i, j) where $j < i$
Thus $\gamma\_rep\ p \subseteq \gamma\_rep\ q$ is not antisymmetric
and thus no partial order.

Intervals

datatype $'a\ extended = Fin\ 'a \mid \infty \mid -\infty$
type_synonym $eint = int\ extended$
type_synonym $eint2 = eint \times eint$

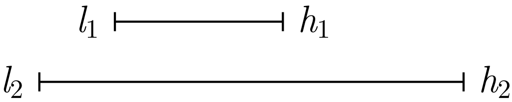
Quotient of $eint2$!

$eq\_ivl :: eint2 \Rightarrow eint2 \Rightarrow bool$
$eq\_ivl\ p_1\ p_2 = (\gamma\_rep\ p_1 = \gamma\_rep\ p_2)$

quotient_type $ivl = eint2\ /\ eq\_ivl$

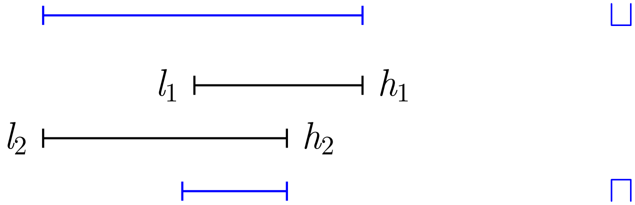
Notation: $[l,h] :: ivl$
Let $\bot = [1,\ 0]$

Partial order on ivl



$$(\bot \leq \_) = \mathit{True}$$
$$(\_ \leq \bot) = \mathit{False}$$
$$([l_1,h_1] \leq [l_2,h_2]) = (l_2 \leq l_1 \wedge h_1 \leq h_2)$$
$$([1,\ 0] \leq [2,\ 3]) \neq (2 \leq 1 \wedge 0 \leq 3)$$

Bounded lattice on ivl



$\bot \sqcup iv = iv$

$iv \sqcup \bot = iv$

$[l_1,\ h_1] \sqcup [l_2,\ h_2] = [min\ l_1\ l_2,\ max\ h_1\ h_2]$

$[1,\ 0] \sqcup [4,\ 5] \neq [1,\ 5]$

$[l_1,\ h_1] \sqcap [l_2,\ h_2] = [max\ l_1\ l_2,\ min\ h_1\ h_2]$

$\top = [-\infty,\ \infty]$

Arithmetic on ivl

$$\perp + iv = \perp$$
$$iv + \perp = \perp$$
$$[l_1, h_1] + [l_2, h_2] = [l_1 + l_2, h_1 + h_2]$$

$$-[l, h] = [-h, -l]$$

$$iv_1 - iv_2 = iv_1 + -iv_2$$

Inverse Analysis of $Plus$

$inv\_plus\_ivl\ iv\ iv_1\ \ iv_2 =$
$(iv_1 \sqcap (iv - iv_2),\ iv_2 \sqcap (iv - iv_1))$

Assume $i_1 \in \gamma\_ivl\ iv_1$, $i_2 \in \gamma\_ivl\ iv_2$, $i_1 + i_2 \in \gamma\_ivl\ iv$
Show $\ i_1 \in \gamma\_ivl\ (iv_1 \sqcap (iv - iv_2))$
$\qquad = \gamma\_ivl\ iv_1 \cap \gamma\_ivl\ (iv - iv_2)$

1. $i_1 \in \gamma\_ivl\ iv_1$ by assumption(1)
2. $i_1 \in \gamma\_ivl\ (iv - iv_2)$
   $= \{i_1.\ \exists\, i{\in}\gamma\_ivl\ iv.\ \exists\, i_2{\in}\gamma\_ivl\ iv_2.\ i_1 = i - i_2\}$
   by assumptions(2,3)

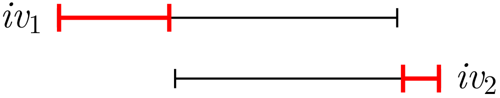
Example: $inv\_plus\_ivl\ [0,\ 4]\ [10,\ 20]\ [-\ 10,\ 0]$
$= ([10,\ 20] \sqcap ([0,\ 4] - [-\ 10,\ 0]), \ldots)$
$= ([10,\ 20] \sqcap [0,\ 14], \ldots) = ([10,\ 14], \ldots)$

Inverse Analysis of $Less$

Case $False$:

Eliminate all points from iv_1 and iv_2 that cannot yield "$\neg\ (Less\ iv_1\ iv_2)$".

Example situation:



$$
\begin{array}{l}
inv\_less\_ivl\ res\ iv_1\ iv_2 = \\
(\textit{if}\ res \\
\ \textit{then}\ (iv_1 \sqcap (below\ iv_2 - [1,\, 1]), \\
\qquad\quad\ iv_2 \sqcap (above\ iv_1 + [1,\, 1])) \\
\ \textit{else}\ (iv_1 \sqcap above\ iv_2,\ iv_2 \sqcap below\ iv_1))
\end{array}
$$

Inverse Analysis of $Less$

$$\begin{array}{l} \mathit{inv\_less\_ivl}\ \mathit{res}\ \mathit{iv}_1\ \mathit{iv}_2 = \\ (\textit{if}\ \mathit{res} \\ \ \textit{then}\ (\mathit{iv}_1 \sqcap (\mathit{below}\ \mathit{iv}_2 - [1,\ 1]), \\ \qquad\quad \mathit{iv}_2 \sqcap (\mathit{above}\ \mathit{iv}_1 + [1,\ 1])) \\ \ \textit{else}\ (\mathit{iv}_1 \sqcap \mathit{above}\ \mathit{iv}_2,\ \mathit{iv}_2 \sqcap \mathit{below}\ \mathit{iv}_1)) \end{array}$$

Example: $\mathit{inv\_less\_ivl}\ \mathit{True}\ [0,\ 20]\ [-\ 5,\ 5]$

$$\begin{array}{l} = ([0,\ 20] \sqcap (\mathit{below}\ [-\ 5,\ 5] - [1,\ 1]), \ldots) \\ = ([0,\ 20] \sqcap ([-\infty,\ 5] - [1,\ 1]), \ldots) \\ = ([0,\ 20] \sqcap [-\infty,\ 4], \ldots) \\ = ([0,\ 4], \ldots) \end{array}$$

Abs_Int2_ivl.thy

The problem

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

Canonical example: intervals

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

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:
  $f(x_i) \leq x_i$
- Widen in each step: $x_{i+1} = x_i \bigtriangledown f(x_i)$
  until a pfp is found.
- We assume
  - $\bigtriangledown$ "extrapolates" its arguments: $x, y \leq x \bigtriangledown y$
  - $\bigtriangledown$ "jumps" far enough to prevent nontermination

Example: Widening on (non-empty) intervals

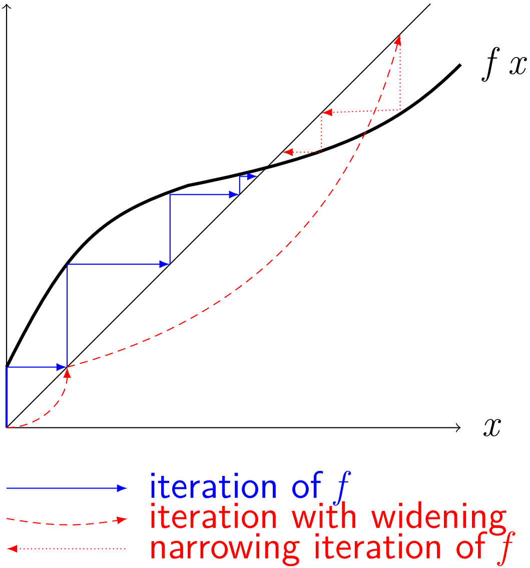
$$[l_1,h_1] \bigtriangledown [l_2,h_2] = [l,h]$$

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

Warning

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

We win termination, we lose precision



f x
x
iteration of f
iteration with widening
narrowing iteration of f

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

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

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

$iter\_widen\ f =$
$while\_option\ (\lambda x.\ \neg\ f\ x \leq x)\ (\lambda x.\ x \bigtriangledown f\ x)$

Correctness (returns pfp): by definition

Abstract interpretation of c:

$$iter\_widen\ (step'\ \top)\ (bot\ c)$$

Interval example

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

Narrowing

Widening returns a (potentially) imprecise pfp p.

If f is monotone, further iteration improves p:

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

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

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

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

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

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

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: $x \triangle f\ x < x$

Example: Narrowing on (non-empty) intervals

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

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

Abstract interpretation with widening & narrowing

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

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

$AI\_wn\ c = pfp\_wn\ (step'\ \top)\ (bot\ c)$

Theorem $AI\_wn\ c = Some\ C \Longrightarrow CS\ c \leq \gamma_c\ C$
Proof as before

Termination

of

$$\mathit{while\_option}\ (\lambda x.\ P\ x)\ (\lambda x.\ g\ x)$$

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:

$$\mathit{Inv}\ x \Longrightarrow P\ x \Longrightarrow m\ x > m(g\ x)$$

Termination of $iter\_widen$

$iter\_widen\ f =$
$while\_option\ (\lambda x.\ \neg\ f\ x \le x)\ (\lambda x.\ x \mathbin{\triangledown} f\ x)$

As before (almost): Assume $m :: {'av} \Rightarrow nat$ and $h :: nat$
such that $m\ x \le h$ and $x \le y \Longrightarrow m\ y \le m\ x$
and additionally $\neg\ y \le x \Longrightarrow m\ (x \mathbin{\triangledown} y) < m\ x$

Define the same functions $m_s/m_o/m_c$ as before.

Termination of $iter\_widen$ on $'a\ st\ option\ acom$:
Lemma $\neg\ C_2 \le C_1 \Longrightarrow m_c\ (C_1 \mathbin{\triangledown} C_2) < m_c\ C_1$
if $top\_on_c\ C_1\ (-\ vars\ C_1)$, $top\_on_c\ C_2\ (-\ vars\ C_2)$
and $strip\ C_1 = strip\ C_2$

Termination of $iter\_narrow$

$iter\_narrow\ f =$
$while\_option\ (\lambda x.\ x \bigtriangleup f\ x < x)\ (\lambda x.\ x \bigtriangleup f\ x)$

Assume $n :: {}'av \Rightarrow nat$ such that
$[\![y \le x;\ x \bigtriangleup y < x]\!] \Longrightarrow n\ (x \bigtriangleup y) < n\ x$

Define $n_s/n_o/n_c$ like $m_s/m_o/m_c$

Termination of $iter\_narrow$ on $'a\ st\ option\ acom$:
Lemma $[\![\ C_2 \le C_1;\ C_1 \bigtriangleup C_2 < C_1\ ]\!] \Longrightarrow$
$n_c\ (C_1 \bigtriangleup C_2) < n_c\ C_1$ if $strip\ C_1 = strip\ C_2$,
$top\_on_c\ C_1\ (-\ vars\ C_1)$ and $top\_on_c\ C_2\ (-\ vars\ C_2)$

Measuring non-empty intervals

$$\begin{aligned} m\ [l,h] = {} & (\mathit{if}\ l = -\infty\ \mathit{then}\ 0\ \mathit{else}\ 1) + \\ & (\mathit{if}\ h = \infty\ \mathit{then}\ 0\ \mathit{else}\ 1) \end{aligned}$$

$$h = 2$$

$$n\ \mathit{ivl} = 2 - m\ \mathit{ivl}$$