Concrete Semantics
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2017-2-8
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:

```
com ::= SKIP
     | string ::= aexp
     | com ;; com
     | IF bexp THEN com ELSE com
     | WHILE bexp DO com
```
Commands

Abstract syntax:

\[
\text{datatype } \text{com} = \begin{align*}
& \text{SKIP} \\
& \text{Assign string aexp} \\
& \text{Seq com com} \\
& \text{If bexp com com} \\
& \text{While bexp com}
\end{align*}
\]
Com.thy
1 IMP Commands

2 Big-Step Semantics

3 Small-Step Semantics
Big-step semantics

Concrete syntax:

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

Intended meaning of $(c, s) \Rightarrow t$:
Command $c$ started in state $s$ terminates in state $t$

“$\Rightarrow$” here not type!
Big-step rules

\[(SKIP, s) \Rightarrow s\]

\[(x ::= a, s) \Rightarrow s(x ::= aval a s)\]

\[\begin{align*}
(c_1, s_1) & \Rightarrow s_2 \\
(c_2, s_2) & \Rightarrow s_3
\end{align*}\]

\[\frac{\begin{align*}
(c_1;.; c_2, s_1) & \Rightarrow s_3
\end{align*}}{}
\]
Big-step rules

\[
\begin{align*}
& bval \ b \ s \quad (c_1, \ s) \Rightarrow t \\
& \quad (IF \ b \ THEN \ c_1 \ ELSE \ c_2, \ s) \Rightarrow t \\
& \quad \neg \ bval \ b \ s \quad (c_2, \ s) \Rightarrow t \\
& \quad (IF \ b \ THEN \ c_1 \ ELSE \ c_2, \ s) \Rightarrow t
\end{align*}
\]
Big-step rules

\[ \neg \text{bval } b \; s \]
\[ \frac{}{(\text{WHILE } b \; \text{DO } c, \; s) \Rightarrow s} \]

\[ \text{bval } b \; s_1 \]
\[ \frac{(c, \; s_1) \Rightarrow s_2 \quad (\text{WHILE } b \; \text{DO } c, \; s_2) \Rightarrow s_3}{(\text{WHILE } b \; \text{DO } c, \; s_1) \Rightarrow s_3} \]
Examples: derivation trees

\[ ("x" ::= N 5; "y" ::= V "x", s) \Rightarrow ? \]

\[ (w, s_i) \Rightarrow ? \]

where

\[ w = \text{WHILE } b \text{ DO } c \]
\[ b = \text{NotEq} (V "x") (N 2) \]
\[ c = "x" ::= \text{Plus} (V "x") (N 1) \]
\[ s_i = s("x" ::= i) \]

\[ \text{NotEq } a_1 a_2 = \]
\[ \text{Not}(\text{And} (\text{Not}(\text{Less } a_1 a_2)) (\text{Not}(\text{Less } a_2 a_1))) \]
Logically speaking

\[(c, s) \Rightarrow t\]

is just infix syntax for

\[\text{big}_\text{step} \ (c,s) \ t\]

where

\[\text{big}_\text{step} ::= \text{com} \times \text{state} \Rightarrow \text{state} \Rightarrow \text{bool}\]

is an inductively defined predicate.
Big_Step.thy

Semantics
Rule inversion

What can we deduce from

- \((\text{SKIP}, s) \Rightarrow t\) ?
- \((x ::= a, s) \Rightarrow t\) ?
- \((c_1;; c_2, s_1) \Rightarrow s_3\) ?
- \((\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \Rightarrow t\) ?
- \((w, s) \Rightarrow t\) where \(w = \text{WHILE } b \text{ DO } c\) ?
Automating rule inversion

Isabelle command \texttt{inductive\_cases} produces theorems that perform rule inversions automatically.
We reformulate the inverted rules. Example:

\[
\begin{align*}
(c_1, c_2, s_1) & \Rightarrow s_3 \\
\exists s_2. (c_1, s_1) & \Rightarrow s_2 \land (c_2, s_2) \Rightarrow s_3
\end{align*}
\]

is logically equivalent to

\[
\begin{align*}
(c_1, c_2, s_1) & \Rightarrow s_3 \\
\bigwedge s_2. [(c_1, s_1) & \Rightarrow s_2; (c_2, s_2) \Rightarrow s_3] & \Rightarrow P
\end{align*}
\]

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 \(\land\)!
The general format: elimination rules

\[ \text{asm} \quad \text{asm}_1 \quad \Rightarrow \quad P \quad \ldots \quad \text{asm}_n \quad \Rightarrow \quad P \]

\[ P \]

(possibly with \( \bigwedge \bar{x} \) in front of the \( \text{asm}_i \Rightarrow P \))

Reading:
To prove a goal \( P \) with assumption \( \text{asm} \),
prove all \( \text{asm}_i \Rightarrow P \)

Example:

\[ F \quad \lor \quad G \quad F \quad \Rightarrow \quad P \quad G \quad \Rightarrow \quad P \]

\[ P \]
Theorems with \textit{elim} attribute are used automatically by \textit{blast}, \textit{fastforce} and \textit{auto}.

Can also be added locally, e.g. \textit{(blast elim: \ldots)}.

Variant: \textit{elim!} applies elim-rules eagerly.
Big_Step.thy

Rule inversion
Command equivalence

Two commands have the same input/output behaviour:

\[ c \sim c' \equiv (\forall s \ t. (c,s) \Rightarrow t \iff (c',s) \Rightarrow t) \]

Example

\[ w \sim w' \]

where

\[ w = \text{WHILE } b \ \text{DO } c \]
\[ w' = \text{IF } b \ \text{THEN } c ;; \ w \ \text{ELSE } \text{SKIP} \]
Equivalence proof

\[(w, s) \Rightarrow t\]

\[\iff\]

\[bval b s \land (\exists s'. (c, s) \Rightarrow s' \land (w, s') \Rightarrow t)\]

\[\lor\]

\[\neg bval b s \land t = s\]

\[\iff\]

\[(w', s) \Rightarrow t\]

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:

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

Needs a formal notion of nontermination to prove it. Could be wrong if we have forgotten a \( \Rightarrow \) rule.
Big-step semantics cannot directly describe
- nonterminating computations,
- parallel computations.

*We need a finer grained semantics!*
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 \(cs \rightarrow cs'\) we say that \(cs\) **reduces** to \(cs'\).
- A configuration \(cs\) is **final** iff \(\not\exists cs'. cs \rightarrow cs'\).
The intention:

\[(\text{SKIP}, \ s) \text{ is final}\]

Why?

\textit{SKIP} is the empty program. Nothing more to be done.
Small-step rules

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

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

$$\begin{align*}
(c_1, s) & \rightarrow (c_1', s') \\
(c_1 ;; c_2, s) & \rightarrow (c_1', ;; c_2, s')
\end{align*}$$
Small-step rules

\[
\begin{align*}
\text{bval } b & \quad s \\
\text{(IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) & \rightarrow (c_1, s) \\
\neg \text{bval } b & \quad s \\
\text{(IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) & \rightarrow (c_2, s)
\end{align*}
\]

\[
\text{(WHILE } b \text{ DO } c, s) \rightarrow (\text{IF } b \text{ THEN } c;; \text{ WHILE } b \text{ DO } c \text{ ELSE SKIP, } s)
\]

**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 \( w = WHILE \ b \ DO \ c \)

\[
\begin{align*}
\begin{array}{ll}
b &= Less \ (V \"x\") \ (N \ 1) \\
c &= \"x\" ::= Plus \ (V \"x\") \ (N \ 1) \\
s_n &= <\"x\" := n>
\end{array}
\end{align*}
\]
Small_Step.thy

Semantics
Are big and small-step semantics equivalent?
Theorem \( cs \Rightarrow t \quad \Rightarrow \quad cs \rightarrow^* (SKIP, t) \)

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

In two cases a lemma is needed:

Lemma

\[
(c_1, s) \rightarrow^* (c_1', s') \quad \Rightarrow \quad (c_1;; c_2, s) \rightarrow^* (c_1';; c_2, s')
\]

Proof by rule induction.
From $\rightarrow^*$ to $\Rightarrow$

**Theorem** $cs \rightarrow^* (\text{SKIP}, t) \implies cs \Rightarrow t$

Proof by rule induction on $cs \rightarrow^* (\text{SKIP}, t)$.

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 \( cs \Rightarrow t \iff cs \rightarrow^* (SKIP, t) \)
Small_Step.thy

Equivalence of big and small
Can execution stop prematurely?
That is, are there any final configs except \((\text{SKIP}, s)\) ?

**Lemma** \(\text{final} (c, s) \implies c = \text{SKIP}\)

We prove the contrapositive

\[ c \neq \text{SKIP} \implies \neg \text{final}(c, s) \]

by induction on \(c\).

- Case \(c_1 ;; c_2\): by case distinction:
  - \(c_1 = \text{SKIP} \implies \neg \text{final} (c_1 ;; c_2, s)\)
  - \(c_1 \neq \text{SKIP} \implies \neg \text{final} (c_1, s) \) (by IH)
    \[\implies \neg \text{final} (c_1 ;; c_2, s)\]
- Remaining cases: trivial or easy
By rule inversion: \((SKIP, s) \rightarrow ct \implies False\)

Together:

**Corollary** \(final (c, s) = (c = SKIP)\)
Infinite executions

⇒ yields final state iff $\rightarrow$ terminates

Lemma \((\exists t. \text{cs} \Rightarrow t) = (\exists \text{cs}'. \text{cs} \rightarrow^{\ast} \text{cs}' \land \text{final} \text{ cs}')\)

Proof: \((\exists t. \text{cs} \Rightarrow t)\)

\[= (\exists t. \text{cs} \rightarrow^{\ast} (\text{SKIP}, t))\]

(by big = small)

\[= (\exists \text{cs}'. \text{cs} \rightarrow^{\ast} \text{cs}' \land \text{final} \text{ cs}')\]

(by final = SKIP)

Equivalent:

⇒ does not yield final state iff $\rightarrow$ does not terminate
May versus Must

→ 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 → path)
- **must** terminate (all → paths terminate)

Therefore: ⇒ 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:

\[\text{datatype } instr = \]
\[\quad \text{LOADI int} \quad \text{load value}\]
\[\quad \text{LOAD vname} \quad \text{load var}\]
\[\quad \text{ADD} \quad \text{add top of stack}\]
\[\quad \text{STORE vname} \quad \text{store var}\]
\[\quad \text{JMP int} \quad \text{jump}\]
\[\quad \text{JMPLESS int} \quad \text{jump if } <\]
\[\quad \text{JMPGE int} \quad \text{jump if } \geq\]
Type synonyms:

\[
\begin{align*}
\text{stack} & \quad = \quad \text{int list} \\
\text{config} & \quad = \quad \text{int} \times \text{state} \times \text{stack}
\end{align*}
\]

Execution of 1 instruction:

\[
iexec :: \text{instr} \Rightarrow \text{config} \Rightarrow \text{config}
\]
**Instruction execution**

\[
\text{iexec instr } (i, s, stk) =
\]

\[
(\text{case instr of LOADI } n \Rightarrow (i + 1, s, n \# stk) \\
\mid \text{LOAD } x \Rightarrow (i + 1, s, s x \# stk) \\
\mid \text{ADD} \Rightarrow (i + 1, s, (\text{hd2 stk} + \text{hd stk}) \# \text{tl2 stk}) \\
\mid \text{STORE } x \Rightarrow (i + 1, s(x := \text{hd stk}), \text{tl stk}) \\
\mid \text{JMP } n \Rightarrow (i + 1 + n, s, stk) \\
\mid \text{JMPLESS } n \Rightarrow \\
\quad (\text{if } \text{hd2 stk} < \text{hd stk} \text{ then } i + 1 + n \text{ else } i + 1, s, \text{tl2 stk}) \\
\mid \text{JMPGE } n \Rightarrow \\
\quad (\text{if } \text{hd stk} \leq \text{hd2 stk} \text{ then } i + 1 + n \text{ else } i + 1, s, \text{tl2 stk}))
\]
Program execution (1 step)

Programs are instruction lists.

Executing one program step:

\[ \text{instr list} \vdash \text{config} \rightarrow \text{config} \]

\[ P \vdash c \rightarrow c' = \]

\[(\exists i \ s \ stk. \]
\[ c = (i, s, stk) \land \]
\[ c' = \text{iexec} (P !! i) (i, s, stk) \land \]
\[ 0 \leq i \land i < \text{size } P) \]

where \( \text{'a list !! int} = \text{nth instruction of list} \)

\( \text{size :: 'a list } \Rightarrow \text{int} = \text{list size as integer} \)
Program execution (∗ steps)

Defined in the usual manner:

\[ P \vdash (pc, s, stk) \rightarrow^\ast (pc', s', stk') \]
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:

\[
acomp\ a \\
\vdash (\emptyset,\ s,\ stk) \rightarrow^* (size\ (acomp\ a),\ s,\ aval\ a\ s\ \#\ stk)
\]

Proof by induction on $a$ (with arbitrary $stk$).

Needs lemmas!
$P \vdash c \rightarrow* c' \implies P @ P' \vdash c \rightarrow* c'$

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

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

$P \vdash c \rightarrow c' \implies P @ P' \vdash c \rightarrow c'$

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

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 $b = And\ (Less\ (V\ "x")\ (V\ "y"))$

$(Not\ (Less\ (V\ "z")\ (V\ "a")))$.

\[\text{bcomp \ b \ False \ 3} = \]

[\text{LOAD } "x",
 \text{LOAD } "y",
 \text{LOAD } "z",
 \text{LOAD } "a",
]

\[ bcomp :: \text{bexp} \Rightarrow \text{bool} \Rightarrow \text{int} \Rightarrow \text{instr list} \]

\[
\begin{align*}
bcomp (Bc \; v) \; f \; n &= (\text{if } v = f \text{ then } [\text{JMP } n] \text{ else } []) \\
bcomp (\text{Not } b) \; f \; n &= bcomp \; b \; (\neg f) \; n \\
bcomp (\text{Less } a_1 \; a_2) \; f \; n &= \\
\text{acomp} \; a_1 @ \\
\text{acomp} \; a_2 @ (\text{if } f \text{ then } [\text{JMPLESS } n] \text{ else } [\text{JMPGE } n]) \\
bcomp (\text{And } b_1 \; b_2) \; f \; n &= \\
\text{let } cb_2 = bcomp \; b_2 \; f \; n; \\
\quad m = \text{if } f \text{ then } \text{size } cb_2 \text{ else } \text{size } cb_2 + n; \\
\quad cb_1 = bcomp \; b_1 \; \text{False } m \\
\text{in } cb_1 @ cb_2
\end{align*}
\]
Correctness of \( \text{bcomp} \)

\[
0 \leq n \implies \text{bcomp } b \ f \ n \\
\vdash (0, s, stk) \rightarrow^* \\
(\text{size } (\text{bcomp } b \ f \ n) + (\text{if } f = \text{bval } b \ s \text{ then } n \text{ else } 0), s, stk)
\]
Compiling \textit{com}

\[
\text{ccomp} :: \text{com} \Rightarrow \text{instr list}
\]

\[
\text{ccomp}\;\text{SKIP} = []
\]

\[
\text{ccomp}\; (x ::= a) = \text{acomp}\; a \oplus [\text{STORE}\; x]
\]

\[
\text{ccomp}\; (c_1 ;; c_2) = \text{ccomp}\; c_1 \oplus \text{ccomp}\; c_2
\]
\begin{equation}
\text{ccomp } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) = \\
\begin{align*}
\text{let } cc_1 &= \text{ccomp } c_1; \ cc_2 = \text{ccomp } c_2; \\
\text{cb} &= \text{bcomp } b \ False \ (\text{size } cc_1 + 1) \\
\text{in } cb \ \text{@} \ cc_1 \ \text{@} \ \text{JMP } (\text{size } cc_2) \neq cc_2
\end{align*}
\end{equation}
\[
ccomp (WHILE \ b \ DO \ c) = \\
\text{let} \ cc = ccomp \ c; \ cb = bcomp \ b \ False \ (size \ cc + 1) \\\ni \ cb @ cc @ [JMP \ (- \ (size \ cb + size \ cc + 1))]\
\]
Correctness of \textit{ccomp}

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

\[(c, s) \Rightarrow t \Rightarrow \text{ccomp}\ c \vdash (0, s, \text{stk}) \rightarrow^* (\text{size (ccomp c), t, stk})\]

Proof by rule induction.
The other direction

We have only shown “⇒”: 

*compiled code simulates source code.*

How about “⇐”: 

*source code simulates compiled code?*

If `ccomp c` with start state `s` produces result `t`, and if(!) `(c, s) ⇒ t'`, then “⇒” 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 $Compiler2$

$$
ccomp \ c \vdash (0, s, stk) \rightarrow^* (size (ccomp \ c), t, stk') \Rightarrow (c, s) \Rightarrow t
$$
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
Remarks on Type Systems
Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed 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.


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

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

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

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

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

How about *completeness*? Remember Rice:

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

Hence there is no (decidable) type system that accepts *all* programs that have a certain semantic property.
Automatic analysis of semantic program properties is necessarily incomplete.
A Typed Version of IMP
Remarks on Type Systems
Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed IMP
### Arithmetic

**Values:**

```plaintext
datatype \( val = Iv \, \text{int} \mid Rv \, \text{real} \)
```

**The state:**

```plaintext
state = vname \( \Rightarrow \) val
```

**Arithmetic expressions:**

```plaintext
datatype \( aexp =
\begin{align*}
Ic \, \text{int} & \mid Rc \, \text{real} & \mid V \, \text{vname} & \mid \text{Plus} \ aexp \ aexp \\
\end{align*}
```

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:

\[ taval :: aexp \Rightarrow \text{state} \Rightarrow \text{val} \Rightarrow \text{bool} \]

\[ taval (\text{Ic } i) \ s \ (\text{Iv } i) \]
\[ taval (\text{Rc } r) \ s \ (\text{Rv } r) \]
\[ taval (\text{V } x) \ s \ (s \ x) \]
\[ \begin{array}{c}
\frac{\begin{array}{c}
taval \ a_1 \ s \ (\text{Iv } i_1) \\
taval \ a_2 \ s \ (\text{Iv } i_2)
\end{array}}{taval \ (\text{Plus } a_1 \ a_2) \ s \ (\text{Iv } (i_1 + i_2))}
\end{array} \]
\[ \begin{array}{c}
\frac{\begin{array}{c}
taval \ a_1 \ s \ (\text{Rv } r_1) \\
taval \ a_2 \ s \ (\text{Rv } r_2)
\end{array}}{taval \ (\text{Plus } a_1 \ a_2) \ s \ (\text{Rv } (r_1 + r_2))}
\end{array} \]
Example: evaluation of $Plus (V "x") (Ic 1)$

If $s "x" = Iv i$:

$$\text{taval} (V "x") s (Iv i) \quad \text{taval} (Ic 1) s (Iv 1)$$

$$\text{taval} (Plus (V "x") (Ic 1)) s (Iv(i + 1))$$

If $s "x" = Rv r$ : then there is no value $v$ such that

$$\text{taval} (Plus (V "x") (Ic 1)) s v.$$
The functional alternative

\[ taval :: aexp \Rightarrow state \Rightarrow val \ option \]

Exercise!
Boolean expressions

Syntax as before. Semantics:

\[ \text{tbval :: bexp } \Rightarrow \text{ state } \Rightarrow \text{ bool } \Rightarrow \text{ bool} \]

\[
\begin{align*}
\text{tbval (Bc v) } s \ v & \quad \text{tbval } b \ s \ bv && \text{tbval (Not b) } s \ (\neg \ bv) \\
\text{tbval } b_1 \ s \ bv_1 & \quad \text{tbval } b_2 \ s \ bv_2 && \text{tbval (And } b_1 \ b_2 \text{) } s \ (bv_1 \land bv_2) \\
\text{taval } a_1 \ s \ (Iv \ i_1) & \quad \text{taval } a_2 \ s \ (Iv \ i_2) && \text{tbval (Less } a_1 \ a_2 \text{) } s \ (i_1 < i_2) \\
\text{taval } a_1 \ s \ (Rv \ r_1) & \quad \text{taval } a_2 \ s \ (Rv \ r_2) && \text{tbval (Less } a_1 \ a_2 \text{) } s \ (r_1 < r_2)
\end{align*}
\]
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.

  \[ \text{big\_step} :: \text{com} \times \text{state} \Rightarrow \text{state option} \Rightarrow \text{bool} \]

  Complicates formalization.

- Small step semantics:
  error = semantics gets stuck
Small step semantics

\[
\begin{align*}
\text{taval } a & \ s & v \\
(x ::= a, s) & \rightarrow (\text{SKIP}, s(x := v))
\end{align*}
\]

\[
\begin{align*}
\text{tbval } b & \ s & \text{True} \\
(IF \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, s) & \rightarrow (c_1, s)
\end{align*}
\]

\[
\begin{align*}
\text{tbval } b & \ s & \text{False} \\
(IF \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, s) & \rightarrow (c_2, s)
\end{align*}
\]

The other rules remain unchanged.
Let \( c = ("x" ::= Plus (V "x") (Ic 1)). \)

- If \( s "x" = Iv i : \)
  \( (c, s) \rightarrow (SKIP, s("x" := Iv (i + 1))) \)

- If \( s "x" = Rv r : \)
  \( (c, s) \not\rightarrow \)
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 \( \text{Plus} \ (V \text{"x"}) \ (V \text{"y"}) \)?

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 environment \( \Gamma \). Standard notation:

\[ \Gamma \vdash e : \tau \]

Read: In the context of \( \Gamma \), \( e \) has type \( \tau \)
The type of an $aexp$

\[
\Gamma \vdash a : \tau \\
\text{tyenv} \vdash aexp : ty
\]

The rules:

\[
\Gamma \vdash Ic \ i : Ity \\
\Gamma \vdash Rc \ r : Rty \\
\Gamma \vdash V \ x : \Gamma \ x \\
\begin{array}{c}
\Gamma \vdash a_1 : \tau \\
\Gamma \vdash a_2 : \tau \\
\end{array}
\hline
\Gamma \vdash \text{Plus} \ a_1 \ a_2 : \tau
\]
Example

\[ \Gamma \vdash \text{Plus} \left( V "x" \right) \left( \text{Plus} \left( V "x" \right) \left( \text{Ic} \ 0 \right) \right) : ? \]

where \( \Gamma "x" = \text{Ity} \).
Well-typed $bexp$

Notation:

\[
\Gamma \vdash b \\
\text{tyenv} \vdash bexp
\]

Read: *In context $\Gamma$, $b$ is well-typed.*
The rules:

\[ \Gamma \vdash Bc \, v \]

\[ \Gamma \vdash b \]

\[ \Gamma \vdash Not \, b \]

\[ \Gamma \vdash b_1 \quad \Gamma \vdash b_2 \]

\[ \Gamma \vdash And \, b_1 \, b_2 \]

\[ \Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau \]

\[ \Gamma \vdash Less \, a_1 \, a_2 \]

Example: \[ \Gamma \vdash Less \, (Ic \, i) \, (Rc \, r) \] does not hold.
Well-typed commands

Notation:

\[ \Gamma \vdash c \]
\[ \text{tyenv} \vdash \text{com} \]

Read: *In context* \( \Gamma \), *c* *is well-typed*. 
The rules:

\[ \Gamma \vdash SKIP \]
\[ \Gamma \vdash a : \Gamma \ x \]
\[ \Gamma \vdash x ::= a \]
\[ \Gamma \vdash c_1 \quad \Gamma \vdash c_2 \]
\[ \Gamma \vdash c_1 ;; c_2 \]
\[ \Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2 \]
\[ \Gamma \vdash IF b THEN c_1 ELSE c_2 \]
\[ \Gamma \vdash b \quad \Gamma \vdash c \]
\[ \Gamma \vdash WHILE b DO c \]
Syntax-directedness

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

- There is exactly one rule for each syntactic construct ($SKIP$, $::=, \ldots$).
- Well-typedness of a term $C\ t_1 \ldots t_n$ depends only on the well-typedness of its subterms $t_1, \ldots, t_n$.

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 \textit{WHILE} depends on the execution of \textit{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  \( ("x" ::= Plus (V "x") (Ic 1), s) \not\rightarrow \)

The state must be well-typed w.r.t. \( \Gamma \).
The type of a value:

\[
\begin{align*}
type (Iv \ i) &= Ity \\
type (Rv \ r) &= Rty
\end{align*}
\]

Well-typed state:

\[
\Gamma \vdash s \leftrightarrow (\forall x. \ type (s \ x) = \Gamma \ x)
\]
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

Progress $\wedge$ Preservation $\implies$ Type safety

**Progress**  Well-typed programs do not get stuck.

**Preservation**  Well-typedness is preserved by reduction.

Preservation: Well-typedness is an *invariant*. 
Progress:

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

Preservation:

\[ [(c, s) \rightarrow (c', s'); \Gamma \vdash c; \Gamma \vdash s] \implies \Gamma \vdash s' \]

\[ [(c, s) \rightarrow (c', s'); \Gamma \vdash c] \implies \Gamma \vdash c' \]

Type soundness:

\[ [(c, s) \rightarrow^* (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq SKIP] \]
\[ \implies \exists cs''. (c', s') \rightarrow cs'' \]
Progress:

\[ [\Gamma \vdash b; \Gamma \vdash s] \implies \exists v. \ tbval b s v \]
Progress:

\[ \left[ \Gamma \vdash a : \tau ; \Gamma \vdash s \right] \implies \exists \; v. \; taval \; a \; s \; v \]

Preservation:

\[ \left[ \Gamma \vdash a : \tau ; \; taval \; a \; s \; v ; \; \Gamma \vdash s \right] \implies \text{type} \; v = \tau \]
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.
A Typed Version of IMP

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 \textit{security/confidentiality levels}.

- Security levels are partially ordered: 
  \( l < l' \) means that \( l \) is less confidential than \( l' \).

- We identify security levels with \textit{nat}.
  Level 0 is public.

- Other popular choices for security levels:
  - only two levels, \textit{high} and \textit{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.
7 Security Type Systems

Secure IMP

A Security Type System
A Type System with Subsumption
A Bottom-Up Type System
Beyond
Security Levels

Security levels:

\texttt{type_synonym} \hspace{0.5em} \texttt{level} = \texttt{nat}

Every variable has a security level:

\texttt{sec :: vname} \Rightarrow \texttt{level}

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

\texttt{sec} \hspace{0.5em} x = \texttt{length} \hspace{0.5em} x
The security level of an expression is the maximal security level of any of its variables.

\[
sec :: \text{aexp} \Rightarrow \text{level}
\]

\[
sec (N \ n) = 0
\]

\[
sec (V \ x) = sec \ x
\]

\[
sec (Plus \ a \ b) = max (sec \ a) \ (sec \ b)
\]
Security Levels on \( bexp \)

\[
sec :: bexp \Rightarrow \text{level}
\]

\[
sec (Bc \; v) = 0
\]

\[
sec (\text{Not} \; b) = sec \; b
\]

\[
sec (\text{And} \; b_1 \; b_2) = \max (sec \; b_1) \; (sec \; b_2)
\]

\[
sec (\text{Less} \; a \; b) = \max (sec \; a) \; (sec \; b)
\]
Security Levels on States

Agreement of states up to a certain level:

\[ s_1 = s_2 \ (\leq l) \equiv \forall x. \ \text{sec} \ x \leq l \rightarrow s_1 \ x = s_2 \ x \]

\[ s_1 = s_2 \ (< l) \equiv \forall x. \ \text{sec} \ x < l \rightarrow s_1 \ x = s_2 \ x \]

Noninterference lemmas for expressions:

\[ s_1 = s_2 \ (\leq l) \quad \text{sec} \ a \leq l \]

\[ \frac{\text{aval} \ a \ s_1 = \text{aval} \ a \ s_2}{bval \ b \ s_1 = bval \ b \ s_2} \]

\[ s_1 = s_2 \ (\leq l) \quad \text{sec} \ b \leq l \]

\[ \frac{\text{aval} \ a \ s_1 = \text{aval} \ a \ s_2}{bval \ b \ s_1 = bval \ b \ s_2} \]
7 Security Type Systems

Secure IMP

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

\[ l \vdash SKIP \]

\[
sec a \leq sec x \quad l \leq sec x
\]

\[ l \vdash x ::= a \]

\[
l \vdash c_1 \quad l \vdash c_2
\]

\[ l \vdash c_1 ;; c_2 \]

\[
max (sec b) \quad l \vdash c_1 \quad max (sec b) \quad l \vdash c_2
\]

\[ l \vdash IF b THEN c_1 ELSE c_2 \]

\[
max (sec b) \quad l \vdash c
\]

\[ l \vdash WHILE b DO c \]
Remark:

\( l \vdash c \) is syntax-directed and executable.
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 \( < l \):

\[
(c, s) \Rightarrow t \quad l \vdash c \\
\frac{}{s = t \ (< l)}
\]

The effect of \( c \) is \textit{confined} to variables of level \( \geq l \).

Proof by \ldots as usual.
Noninterference

\[(c, s) \Rightarrow s' \quad (c, t) \Rightarrow t' \quad 0 \vdash c \quad s = t \quad (\leq l)\]

\[s' = t' \quad (\leq l)\]

Proof by \ldots as usual.
Security Type Systems

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond
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

\[
\frac{l \vdash' c \quad l' \leq l}{l' \vdash' c}
\]

together with one rule per construct:
\[
\begin{align*}
l & \vdash \ ' \ \text{SKIP} \\
\text{sec} \ a & \leq \text{sec} \ x & l & \leq \text{sec} \ x \\
l & \vdash \ ' \ x ::= \ a \\
l & \vdash \ ' \ c_1 & l & \vdash \ ' \ c_2 \\
l & \vdash \ ' \ c_1 ;; \ c_2 \\
\text{sec} \ b & \leq \ l & l & \vdash \ ' \ c_1 & l & \vdash \ ' \ c_2 \\
l & \vdash \ ' \ \text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \\
\text{sec} \ b & \leq \ l & l & \vdash \ ' \ c \\
l & \vdash \ ' \ \text{WHILE} \ b \ \text{DO} \ c
\end{align*}
\]
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 $\vdash'$

$$l \vdash c \iff l \vdash' c$$

Proof by induction. Use subsumption directly below \textit{IF} and \textit{WHILE}.

$$l \vdash' c \iff l \vdash c$$

Proof by induction. Subsumption already a lemma for $\vdash$. 
7 Security Type Systems

Secure IMP
A Security Type System
A Type System with Subsumption
A Bottom-Up Type System
Beyond
• Systems \( l \vdash c \) and \( l \vdash' c \) are top-down: level \( l \) comes from the context and is checked at ::= commands.

• System \( \vdash c : l \) is bottom-up: \( l \) is the minimal level of any variable assigned in \( c \) and is checked at \( IF \) and \( WHILE \) commands.
\[ \vdash SKIP : l \]

\[ sec \ a \leq sec \ x \]
\[ \vdash x ::= a : sec \ x \]

\[ \vdash c_1 : l_1 \quad \vdash c_2 : l_2 \]
\[ \vdash c_1 ; ; c_2 : min \ l_1 \ l_2 \]

\[ sec \ b \leq min \ l_1 \ l_2 \quad \vdash c_1 : l_1 \quad \vdash c_2 : l_2 \]
\[ \vdash IF \ b \ THEN \ c_1 \ ELSE \ c_2 : min \ l_1 \ l_2 \]

\[ sec \ b \leq l \quad \vdash c : l \]
\[ \vdash WHILE \ b \ DO \ c : l \]
Equivalence of $\vdash :$ and $\vdash'$

$$\vdash c : l \implies l \vdash' c$$

Proof by induction.

$$l \vdash' c \implies \vdash c : l$$

Nitpick: $0 \vdash' "x" ::= N 1$ but not $\vdash "x" ::= N 1 : 0$

$$l \vdash' c \implies \exists l' \geq l. \vdash c : l'$$

Proof by induction.
7 Security Type Systems

Secure IMP
A Security Type System
A Type System with Subsumption
A Bottom-Up Type System
Beyond
Does noninterference really guarantee absence of information flow?

\[
\begin{align*}
(c, s) &\Rightarrow s' \\
(c, t) &\Rightarrow t' \\
0 &\vdash c \\
\frac{s = t (\leq l)}{s' = t' (\leq l)}
\end{align*}
\]

Beware of covert channels!

\[
0 \vdash \textit{WHILE} \textit{Less} (V "x") (N 1) \textit{DO SKIP}
\]
A drastic solution:

**WHILE**-conditions must not depend on confidential data.

New typing rule:

\[
\text{sec } b = 0 \quad 0 \vdash c \\
\hline
0 \vdash \text{WHILE } b \text{ DO } c
\]

Now provable:

\[
(c, s) \Rightarrow s' \quad 0 \vdash c \quad s = t \ (\leq l) \\
\hline
\exists t'. \ (c, t) \Rightarrow t' \land s' = t' \ (\leq l)
\]
Further extensions

- Time
- Probability
- Quantitative analysis
- More programming language features:
  - exceptions
  - concurrency
  - OO
  - …
The inventors of security type systems are Volpano and Smith.

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 $z := x$; $z := z + 1$
We do not analyze boolean expressions to determine program execution.
Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis
Initialization Sensitive Semantics
Theory \( \textit{Vars} \) provides an overloaded function \( \textit{vars} \):

\[
\textit{vars} :: \textit{aexp} \Rightarrow \textit{vname} \text{ set}
\]

\[
\textit{vars} (N \ n) = \{\}
\]

\[
\textit{vars} (V \ x) = \{x\}
\]

\[
\textit{vars} (\text{Plus} \ a_1 \ a_2) = \textit{vars} \ a_1 \cup \textit{vars} \ a_2
\]

\[
\textit{vars} :: \textit{bexp} \Rightarrow \textit{vname} \text{ set}
\]

\[
\textit{vars} (Bc \ v) = \{\}
\]

\[
\textit{vars} (\text{Not} \ b) = \textit{vars} \ b
\]

\[
\textit{vars} (\text{And} \ b_1 \ b_2) = \textit{vars} \ b_1 \cup \textit{vars} \ b_2
\]

\[
\textit{vars} (\text{Less} \ a_1 \ a_2) = \textit{vars} \ a_1 \cup \textit{vars} \ a_2
\]
Vars.thy
Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics
Modified example from the JLS:

$$\text{Variable } x \text{ is definitely initialized after } SKIP \iff x \text{ is definitely initialized before } SKIP.$$ 

Similar statements for each language construct.
\[ D :: vname \text{ set} \Rightarrow \text{ com} \Rightarrow vname \text{ set} \Rightarrow \text{ bool} \]

\[ D \ A \ c \ A' \] 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

vars a ⊆ A

D A (x ::= a) (insert x A)

D A₁ c₁ A₂ D A₂ c₂ A₃

D A₁ (c₁;; c₂) A₃

vars b ⊆ A D A c₁ A₁ D A c₂ A₂

D A (IF b THEN c₁ ELSE c₂) (A₁ ∩ A₂)

vars b ⊆ A D A c A'

D A (WHILE b DO c) A
Correctness of $D$

- Things can go wrong: execution may access uninitialized variable.
  \[ \Rightarrow \text{We need a new, finer-grained semantics.} \]

- 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.
Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics
\textit{state} = \textit{vname} \Rightarrow \textit{val option}

where

\textbf{datatype} \ 'a option = \textit{None} | \textit{Some} 'a

Notation: \( s(x \mapsto y) \) means \( s(x := \textit{Some} y) \)

Definition: \( \textit{dom} s = \{ a. \ s \ a \neq \textit{None} \} \)
Expression evaluation

\[ \text{aval :: aexp} \to \text{state} \to \text{val option} \]

\[ \text{aval (N i) } s = \text{Some } i \]

\[ \text{aval (V x) } s = s \ x \]

\[ \text{aval (Plus } a_1 \ a_2 \text{) } s = \]

\[ \text{(case } \text{aval } a_1 \ s, \text{ aval } a_2 \ s \text{ of} \]

\[ (\text{Some } i_1, \text{ Some } i_2) \Rightarrow \text{Some}(i_1+i_2) \]

\[ | \_ \Rightarrow \text{None} \) \]
bval :: bexp ⇒ state ⇒ bool option

bval (Bc v) s = Some v

bval (Not b) s =

(case bval b s of None ⇒ None
 | Some bv ⇒ Some (¬ bv))

bval (And b₁ b₂) s =

(case (bval b₁ s, bval b₂ s) of
  (Some bv₁, Some bv₂) ⇒ Some(bv₁ ∧ bv₂)
  | _ ⇒ None)

bval (Less a₁ a₂) s =

(case (aval a₁ s, aval a₂ s) of
  (Some i₁, Some i₂) ⇒ Some(i₁ < i₂)
  | _ ⇒ None)
Big step semantics

\[(\text{com}, \text{state}) \Rightarrow \text{state option}\]

A small complication:

\[
(c_1, s_1) \Rightarrow \text{Some } s_2 \quad (c_2, s_2) \Rightarrow s \\
\hline
(c_1;; c_2, s_1) \Rightarrow s
\]

\[
(c_1, s_1) \Rightarrow \text{None} \\
(c_1;; c_2, s_1) \Rightarrow \text{None}
\]

More convenient, because compositional:

\[(\text{com}, \text{state option}) \Rightarrow \text{state option}\]
Error \((None)\) propagates:

\[(c, None) \Rightarrow None\]

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

\[(SKIP, s) \Rightarrow s\]

\[aval a s = Some i \]

\[(x ::= a, Some s) \Rightarrow Some(s(x \mapsto i))\]

\[aval a s = None \]

\[(x ::= a, Some s) \Rightarrow None\]

\[(c_1, s_1) \Rightarrow s_2 \quad (c_2, s_2) \Rightarrow s_3\]

\[(c_1;; c_2, s_1) \Rightarrow s_3\]
\[
\begin{align*}
\text{bval } b \ s &= \text{Some True} \quad (c_1, \text{Some } s) \Rightarrow s' \\
(IF \ b \ \text{THEN } c_1 \ \text{ELSE } c_2, \text{Some } s) &\Rightarrow s'
\end{align*}
\]

\[
\begin{align*}
\text{bval } b \ s &= \text{Some False} \quad (c_2, \text{Some } s) \Rightarrow s' \\
(IF \ b \ \text{THEN } c_1 \ \text{ELSE } c_2, \text{Some } s) &\Rightarrow s'
\end{align*}
\]

\[
\begin{align*}
\text{bval } b \ s &= \text{None} \\
(IF \ b \ \text{THEN } c_1 \ \text{ELSE } c_2, \text{Some } s) &\Rightarrow \text{None}
\end{align*}
\]
\[
\begin{align*}
\text{bval b s} &= \text{Some False} \\
\implies (\text{WHILE b DO c, Some s}) &\Rightarrow \text{Some s}
\end{align*}
\]

\[
\begin{align*}
\text{bval b s} &= \text{Some True} \\
(c, \text{Some s}) &\Rightarrow s' \quad (\text{WHILE b DO c, s'}) &\Rightarrow s'' \\
\implies (\text{WHILE b DO c, Some s}) &\Rightarrow s''
\end{align*}
\]

\[
\begin{align*}
\text{bval b s} &= \text{None} \\
\implies (\text{WHILE b DO c, Some s}) &\Rightarrow \text{None}
\end{align*}
\]
Correctness of $D$ w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

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

We need to prove a generalized statement:

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

By rule induction on $(c, \text{Some } s) \Rightarrow s'$. 
Proof needs some easy lemmas:

\[ \text{vars } a \subseteq \text{dom } s \implies \exists i. \text{aval } a s = \text{Some } i \]

\[ \text{vars } b \subseteq \text{dom } s \implies \exists bv. \text{bval } b s = \text{Some } bv \]

\[ D \ A \ c \ A' \implies A \subseteq A' \]
8 Definite Initialization Analysis

9 Live Variable Analysis
Motivation

Consider the following program:

\[
\begin{align*}
    x & := y + 1; \\
    y & := y + 2; \\
    x & := y + 3
\end{align*}
\]

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 \( x \) \textit{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 \textit{dead} or \textit{live}?
\[
x := 0 \quad \text{😊}
y := x; y := 0; x := 0 \quad \text{😊}
\text{WHILE } b \text{ DO } y := x; x := 1 \quad \text{😊}
\]
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

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

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

$L\ SKIP\ X = X$

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

$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

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

$\text{vars } b \cup L\ c_1\ X \cup L\ c_2\ X$

Example:

$L\ ("y" ::= V\ "z";; \ "x" ::= Plus\ (V\ "y")\ (V\ "z"))$

$\{"x"\} = \{"z"\}$
WHILE \( b \) DO \( c \)

\( L \ w \ X \) must satisfy

- \( \text{vars } b \subseteq L \ w \ X \) (evaluation of \( b \))
- \( X \subseteq L \ w \ X \) (exit)
- \( L \ c \ (L \ w \ X) \subseteq L \ w \ X \) (execution of \( c \))
We define

\[
L \left( \textit{WHILE} \ b \ \textit{DO} \ c \right) \ X = \texttt{vars} \ b \cup X \cup L \ c \ X
\]

\[\Rightarrow\]

\[
\texttt{vars} \ b \subseteq L \ w \ X \quad \checkmark
\]

\[
X \subseteq L \ w \ X \quad \checkmark
\]

\[
L \ c \ (L \ w \ X) \subseteq L \ w \ X \quad ?
\]
\[ L \text{ SKIP } X = X \]
\[ L (x ::= a) X = \text{vars } a \cup (X - \{x\}) \]
\[ L (c_1;;c_2) X = L c_1 (L c_2 X) \]
\[ L (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) X = \text{vars } b \cup L c_1 X \cup L c_2 X \]
\[ L (\text{WHILE } b \text{ DO } c) X = \text{vars } b \cup X \cup L c X \]

Example:
\[ L (\text{WHILE } \text{Less}(V "x") (V "x") \text{ DO } "y" ::= V "z") \]
\[ \{"x"\} = \{"x","z"\} \]
Gen/kill analyses

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

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

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

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

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

\[
\text{kill} :: \text{com} \Rightarrow \text{vname set}
\]

\[
\begin{align*}
\text{kill } \text{SKIP} & = \{\}\; \\
\text{kill } (x ::= a) & = \{x\} \\
\text{kill } (c_1 ;; c_2) & = \text{kill } c_1 \cup \text{kill } c_2 \\
\text{kill } (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) & = \text{kill } c_1 \cap \text{kill } c_2 \\
\text{kill } (\text{WHILE } b \text{ DO } c) & = \{\}\;
\end{align*}
\]
\textit{gen} :: \textit{com} \Rightarrow \textit{vname set}

\begin{align*}
gen \text{ \textit{SKIP}} & = \emptyset \\
gen \ (x ::= a) & = \text{vars } a \\
gen \ (c_1 ; ; c_2) & = gen \ c_1 \cup (gen \ c_2 - \text{kill } c_1) \\
gen \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) & = \\
& \quad \text{vars } b \cup \gen \ c_1 \cup \gen \ c_2 \\
gen \ (\text{WHILE } b \ \text{DO } c) & = \text{vars } b \cup \gen \ c
\end{align*}
\[ L \ c \ X = \text{gen} \ c \cup (X - \text{kill} \ c) \]

Proof by induction on \( c \).

\[ \iff \]

\[ L \ c \ (L \ w \ X) \subseteq L \ w \ X \]
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 \( A_c X = X - \text{kill } c \cup \text{gen } c \):

\[
\begin{align*}
\text{gen } \text{SKIP} & = \{ \} \\
\text{gen } (x ::= a) & = \{ x \} \\
\text{gen } (c_1;; c_2) & = \text{gen } c_1 \cup \text{gen } c_2 \\
\text{gen } (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) & = \text{gen } c_1 \cap \text{gen } c_2 \\
\text{gen } (\text{WHILE } b \text{ DO } c) & = \{ \} \\
\text{kill } c & = \{ \}
\end{align*}
\]
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 \(L \cap X\).*

Put differently:

*If two initial states agree on \(L \cap X\) then the corresponding final states agree on \(X\).*
Equality on

An abbreviation:

\[ f = g \text{ on } X \equiv \forall x \in X. f(x) = g(x) \]

Two easy theorems (in theory \( Vars \)):

\[
\begin{align*}
  &s_1 = s_2 \text{ on vars } a \implies \text{aval } a \ s_1 = \text{aval } a \ s_2 \\
  &s_1 = s_2 \text{ on vars } b \implies \text{bval } b \ s_1 = \text{bval } b \ s_2
\end{align*}
\]
Correctness of $L$

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

Proof by rule induction.
For the two \textit{WHILE} cases we do not need the definition of $L \ w$ but only the characteristic property

$$\text{vars } b \cup X \cup L \ c \ (L \ w \ X) \subseteq L \ w \ X$$
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 $\text{vars } b \cup X \cup L_c (L_w X) \subseteq L_w X$.

Follows easily from $L_c X = \text{gen } c \cup (X - \text{kill } c)$:

$\text{vars } b \cup X \cup L_c P \subseteq P \implies L (\text{WHILE } b \text{ DO } c) X \subseteq P$
Live Variable Analysis

Correctness of $L$

Dead Variable Elimination

True Liveness

Comparisons
Bury all assignments to dead variables:

\[
\text{bury} :: \text{com} \Rightarrow \text{vname set} \Rightarrow \text{com}
\]

\[
\begin{align*}
\text{bury } \text{SKIP } X &= \text{SKIP} \\
\text{bury } (x ::= a) \ X &= \text{if } x \in X \ \text{then } x ::= a \ \text{else } \text{SKIP} \\
\text{bury } (c_1;; c_2) \ X &= \text{bury } c_1 (L c_2 X);; \text{bury } c_2 X \\
\text{bury } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ X &= \text{IF } b \ \text{THEN } \text{bury } c_1 X \ \text{ELSE } \text{bury } c_2 X \\
\text{bury } (\text{WHILE } b \ \text{DO } c) \ X &= \text{WHILE } b \ \text{DO } \text{bury } c (L (\text{WHILE } b \ \text{DO } c) X)
\end{align*}
\]
Correctness of $bury$

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

where $UNIV$ is the set of all variables.

The two directions need to be proved separately.
\[(c, s) \Rightarrow s' \quad \Rightarrow \quad (\text{bury } c \text{ UNIV}, s) \Rightarrow s'\]

Follows from generalized statement:

\text{If } (c, s) \Rightarrow s' \text{ and } s = t \text{ on } L \ c \ X \text{ then } \exists t'. \ (\text{bury } c \ X, t) \Rightarrow t' \land s' = t' \text{ on } X.

Proof by rule induction, like for correctness of \( L \).
\[(bury\ c\ UNIV, s) \Rightarrow s' \iff (c, s) \Rightarrow s'\]

Follows from generalized statement:

*If* \((bury\ c\ X, s) \Rightarrow s'\) *and* \(s = t\) *on* \(L\ c\ X\)

*then* \(\exists t'. (c, t) \Rightarrow t' \land s' = t'\) *on* \(X\).

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

\[(bc_1; ; \ bc_2 = bury\ c\ X) = \]

\[(\exists\ c_1\ c_2.\]

\(c = c_1; ; \ c_2 \land\)

\(bc_2 = bury\ c_2\ X \land\ bc_1 = bury\ c_1\ (L\ c_2\ X))\]
Live Variable Analysis

Correctness of $L$

Dead Variable Elimination

True Liveness

Comparisons
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 \( \tau \), eg \( \subseteq \) on sets.

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

If \( x \leq y \Rightarrow f \ x \leq f \ y \) for all \( x, y \), then \( f \) is monotone.
Remember the specification of $L_w$:

$$\text{vars } b \cup X \cup L_c \ (L_w X) \subseteq L_w X$$

This is the same as saying that $L_w X$ should be a pfp of 

$$\lambda P. \text{vars } b \cup X \cup L_c \ P$$

and in particular of $L_c$. 
True liveness

\[ L \left( "x" ::= V "y" \right) \{ \} = \{ "y" \} \]

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

Problem: \( L \left( x ::= a \right) X = \text{vars } a \cup (X - \{ x \}) \)

Better:

\[ L \left( x ::= a \right) X = \]
\( \text{(if } x \in X \text{ then vars } a \cup (X - \{ x \}) \text{ else } X) \)

But then

\[ L \left( \text{WHILE } b \text{ DO } c \right) X = \text{vars } b \cup X \cup L \ c \ X \]

is not correct anymore.
\[ L \ (x ::= \ a) \ X = \]
\[(\text{if} \ x \in X \ \text{then} \ \text{vars} \ a \cup (X - \{x\}) \ \text{else} \ X)\]

\[ L \ (\text{WHILE} \ b \ \text{DO} \ c) \ X = \text{vars} \ b \cup X \cup L \ c \ X \]

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

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

\[ \{x\} \ y ::= V \ x \ \{y\} \ x ::= V \ z \ \{y\} \]

\[ \implies L \ w \ \{y\} = \{y\} \cup \{y\} \cup \{x\} \]
\[ b = \text{Less} \ (N \ 0) \ (V \ y) \]
\[ c = y ::= V \ x; \ x ::= V \ z \]

\[ L \ w \ \{ y\} = \{ x, \ y\} \text{ is not a pfp of } L \ c: \]
\[ \{ x, \ z\} \ y ::= V \ x \ \{ y, \ z\} \ x ::= V \ z \ \{ x, \ y\} \]
\[ L \ c \ \{ x, \ y\} = \{ x, \ z\} \not\subseteq \{ x, \ y\} \]
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 \( f :: \tau \text{ set} \Rightarrow \tau \text{ set} \). If \( f \) is monotone (\( X \subseteq Y \implies f(X) \subseteq f(Y) \)), then

\[
\text{lfp}(f) := \bigcap \{ P \mid f(P) \subseteq P \}
\]

is the least pre-fixpoint and least fixpoint of \( f \).
Proof of Knaster-Tarski

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

Proof  
\begin{itemize}
  \item \( f(\text{lfp } f) \subseteq \text{lfp } f \)
  \item \( \text{lfp } f \) is the least pre-fixpoint of \( f \)
\end{itemize}

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  
\begin{itemize}
  \item \( f p \leq p \implies f p = p \)
  \item \( p \) is the least fixpoint
Definition of $L$

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

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

**Lemma** $L \ c$ is monotone.

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

**Corollary** $f_w$ is monotone.
Computation of $lfp$

**Theorem** Let $f : \tau \text{ set} \Rightarrow \tau \text{ set}$. If

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

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

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

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

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

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

**Lemma** $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on $c$

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

**Corollary** $P \subseteq V_w \implies f_w \ P \subseteq V_w$

Hence $f_w^k \{\}$ stabilizes for some $k \leq |V_w|$.

More precisely: $k \leq |\text{vars } c| + 1$

because $f_w \{\} \supseteq \text{vars } b \cup X$. 
Example

Let $w = \text{WHILE } b \text{ DO } c$
where $b = \text{Less } (N \ 0) \ (V \ y)$
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`:

```plaintext
while :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
while b f s = (if b s then while b f (f s) else s)
```

**Lemma** Let $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$. If

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

then $lfp\ f = \text{while } (\lambda X. f X \neq X)\ f\ {}$
Limiting the number of iterations

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

\[
Lb \ w \ X = \begin{cases} 
  g^i_w \ \{\} & \text{if } g^{i+1}_w \ \{\} = g^i_w \ \{\} \text{ for some } i < k \\
  V_w & \text{otherwise}
\end{cases}
\]

where \( g_w \ P = \text{vars } b \cup X \cup Lb \ c \ P \)

**Theorem** \( L \ c \ X \subseteq Lb \ c \ X \)

**Proof** by induction on \( c \). In the \textit{WHILE} case:

If \( Lb \ w \ X = g^i_w \ \{\} \):

\[
\forall P. \ L \ c \ P \subseteq Lb \ c \ P \quad (\text{IH}) \implies \forall P. \ f_w \ P \subseteq g_w \ P \implies f_w(g^i_w \ \{\}) = g_w(g^i_w \ \{\}) = g^i_w \ \{\}
\]

\[
\implies L \ w \ X = \text{lfp } f_w \subseteq g^i_w \ \{\} = Lb \ w \ X
\]

If \( Lb \ w \ X = V_w \):

\( L \ w \ X \subseteq V_w \) (by Lemma)
9 Live Variable Analysis

Correctness of $L$

Dead Variable Elimination

True Liveness

Comparisons
Comparison of analyses

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

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

Program representation:

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

- We analyse structured programs. Application: source-level program optimization.
Chapter 11

Denotational Semantics
A Relational Denotational Semantics of IMP

Continuity
What is it?

A denotational semantics maps syntax to semantics:

\[ D :\text{ syntax} \Rightarrow \text{ meaning} \]

Examples:

\[ \text{aval} : \text{ aexp} \Rightarrow (\text{state} \Rightarrow \text{val}) \]
\[ \text{Big\_step} : \text{ com} \Rightarrow (\text{state} \times \text{state}) \text{ set} \]

\(D\) must be defined by primitive recursion over the syntax

\[ D (C \ t_1 \ldots \ t_n) = \ldots (D \ t_1) \ldots (D \ t_n) \ldots \]

Fake: \[\text{Big\_step} \ c = \{(s,t). \ (c,s) \Rightarrow t\}\]
More abstract:
  operational: How to execute it
  denotational: What does it mean

Simpler proof principles:
  operational: relational, rule induction
  denotational: equational, structural induction
A Relational Denotational Semantics of IMP

Continuity
Relations

\[ Id :: (\mathcal{A} \times \mathcal{A}) \text{ set} \]
\[ Id = \{ p. \ \exists x. \ p = (x, x) \} \]

\[ \text{op } O :: (\mathcal{A} \times \mathcal{B}) \text{ set } \Rightarrow (\mathcal{B} \times \mathcal{C}) \text{ set } \Rightarrow (\mathcal{A} \times \mathcal{C}) \text{ set} \]
\[ r \ O \ s = \{ (x, z). \ \exists y. \ (x, y) \in r \land (y, z) \in s \} \]
\( D :: \) \( \text{com} \Rightarrow \text{com\_den} \)

type\_synonym \( \text{com\_den} = (\text{state} \times \text{state}) \text{ set} \)

\[
D \text{ SKIP} = \text{Id}
\]

\[
D (x ::= a) = \{ (s, t). \ t = s(x ::= \text{aval} a s) \}
\]

\[
D (c_1 ;; c_2) = D c_1 O D c_2
\]

\[
D (\text{IF } b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2) = \{ (s, t). \ \text{if } \text{bval} b s \ \text{then} \ (s, t) \in D c_1 \ \text{else} \ (s, t) \in D c_2 \}
\]
Example

Let \( c_1 = "x" ::= N \ 0 \)
\( c_2 = "y" ::= V "x":\)

\[
D \ c_1 = \{ (s_1,s_2). \ s_2 = s_1("x" := 0) \}
\]

\[
D \ c_2 = \{ (s_2,s_3). \ s_3 = s_2("y" := s_2 "x") \}
\]

\[
D \ (c_1;;c_2) = \{ (s_1,s_3). \ s_3 = s_1("x" := 0, "y" := 0) \}
\]
\[ D \ (WHILE \ b \ DO \ c) = ? \]

Wanted:

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

Problem: not a denotational definition
not allowed by Isabelle

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

General principle:

\[ x \text{ is a solution of } x = f(x) \iff x \text{ is a fixpoint of } f \]

Define \( D\ w \) as the least fixpoint of a suitable \( f \)
\[ D \ w = \{(s, t). \ if \ bval \ b \ s \ then \ (s, t) \in \ D \ c \ O \ D \ w \ else \ s = t\}\]

\[ W :: (state \Rightarrow \text{bool}) \Rightarrow \text{com\_den} \Rightarrow (\text{com\_den} \Rightarrow \text{com\_den}) \]

\[ W \ db \ dc = \]

\[ (\lambda dw. \{(s, t). \ if \ db \ s \ then \ (s, t) \in \ dc \ O \ dw \ else \ s = t\}) \]

**Lemma** \( W \ db \ dc \) is monotone.
We define

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

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

$$D \ w = \text{lfp} \ f = f \ (\text{lfp} \ f) = W \ (bval \ v) \ (D \ c) \ (D \ w)$$

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

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

\[ w = \text{WHILE } Bc \text{ True DO SKIP} \]

Then

\[
W (bval (Bc True)) (D SKIP) \\
= W (\lambda s. \text{True}) \text{ Id} \\
= \lambda dw. \{(s, t). (s, t) \in \text{Id O dw}\} \\
= \lambda dw. \text{dw}
\]

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 \( w = \text{WHILE} \ b \ DO \ c \).

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

\[ D \, w \]

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

\[ = D \ (IF \ b \ THEN \ c\;;\; w \ ELSE \ SKIP) \]
Equivalence of denotational and big-step semantics

Lemma \((c, s) \Rightarrow t \iff (s, t) \in D_c\)

Proof by rule induction

Lemma \((s, t) \in D_c \iff (s, t) \in Big_{-}step \ c\)

Proof by induction on \(c\)

Corollary \((s, t) \in D_c \iff (c, s) \Rightarrow t\)
A Relational Denotational Semantics of IMP

Continuity
Chains and continuity

**Definition**

\[
\text{chain :: (nat } \Rightarrow \text{'a set) } \Rightarrow \text{ bool}
\]

\[
\text{chain } S = (\forall i. S \ i \subseteq S \ (\text{Suc } i))
\]

**Definition (Continuous)**

\[
\text{cont :: ('a set } \Rightarrow \text{'b set) } \Rightarrow \text{ bool}
\]

\[
\text{cont } f = (\forall S. \text{chain } S \xrightarrow{} f \ (\bigcup_n S \ n) = (\bigcup_n f \ (S \ n)))
\]

**Lemma** \(\text{cont } f \xrightarrow{} \text{ mono } f\)
Kleene fixpoint theorem

**Theorem** \( \text{cont } f \iff \text{lfp } f = (\bigcup_n f^n \{\}) \)
**Application to semantics**

**Lemma** \( W \, db \, dc \) is continuous.

**Example**

\[ \begin{align*}
\text{WHILE } x \neq 0 \text{ DO } x & := x - 1 \\
\text{Semantics: } & \{(s, t). \ 0 \leq s \ "x" \wedge t = s("x" := 0)\}
\end{align*} \]

Let \( f = W \, db \, dc \)

where \( db = bval \ b = (\lambda s. \ s \ "x" \neq 0) \)

\( dc = D \ c = \{(s, t). \ t = s("x" := s \ "x" - 1)\} \)
A proof of determinism

\[\text{single_valued } r = (\forall x y z. (x, y) \in r \land (x, z) \in r \rightarrow y = z)\]

**Lemma** If \( f :: \text{com\_den} \Rightarrow \text{com\_den} \) is continuous and preserves single-valuedness then \( \text{lfp } f \) is single-valued.

**Lemma** \( \text{single\_valued } (D c) \)
Chapter 12

Hoare Logic
12 Partial Correctness

13 Verification Conditions

14 Total Correctness
12 Partial Correctness

13 Verification Conditions

14 Total Correctness
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

```
wsum ≡
WHILE Less (N 0) (V "x")
DO ("y" ::= Plus (V "y") (V "x");;
   "x" ::= Plus (V "x") (N (− 1)))
```

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 = (if i ≤ 0 then 0 else sum (i − 1) + i)
```
A proof via operational semantics

Theorem:

\[
("y" ::= N 0;; wsum, s) \Rightarrow t \Rightarrow t "y" = sum (s "x")
\]

Required Lemma:

\[
(wsum, s) \Rightarrow t \Rightarrow t "y" = s "y" + sum (s "x")
\]

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.
Partial Correctness

Introduction

The Syntactic Approach

The Semantic Approach

Soundness and Completeness
This is the standard approach. Formulas are syntactic objects. Everything is very concrete and simple. But complex to formalize. Hence we soon move to a semantic view of formulas. Reason for introduction of syntactic approach: didactic.

For now, we work with a (syntactically) simplified version of IMP.
Hoare Logic reasons about *Hoare triples* \( \{ P \} \ c \ \{ Q \} \)

where

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

Informal example:

\[
\{ x = 41 \} \ x := x + 1 \ \{ x = 42 \}
\]

Terminology: \( P \) and \( Q \) are called *assertions*. 
Examples

\( \{x = 5\} \quad ? \quad \{x = 10\} \)

\( \{True\} \quad ? \quad \{x = 10\} \)

\( \{x = y\} \quad ? \quad \{x \neq y\} \)

Boundary cases:

\( \{True\} \quad ? \quad \{True\} \)

\( \{True\} \quad ? \quad \{False\} \)

\( \{False\} \quad ? \quad \{Q\} \)
The rules of Hoare Logic

\[{P} \ \text{SKIP} \ {P}\]

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

Notation: \( Q[a/x] \) means “\( Q \) with \( a \) substituted for \( x \)”.  

Examples: \[
\begin{align*}
    \{} & x := 5 \quad \{x = 5\} \\
    \{} & x := x + 5 \quad \{x = 5\} \\
    \{} & x := 2 \times (x + 5) \quad \{x > 20\}
\end{align*}
\]

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

\[
\begin{align*}
\{P_1\} & \quad c_1 \quad \{P_2\} \quad \{P_2\} & \quad c_2 \quad \{P_3\} \\
\{P_1\} \quad c_1; c_2 \quad \{P_3\}
\end{align*}
\]

\[
\begin{align*}
\{P \land b\} & \quad c_1 \quad \{Q\} \quad \{P \land \neg b\} & \quad c_2 \quad \{Q\} \\
\{P\} \quad IF \ b \ THEN \ c_1 \ Quad ELSE \ c_2 \quad \{Q\}
\end{align*}
\]

\[
\begin{align*}
\{P \land b\} & \quad c \quad \{P\} \\
\{P\} \quad WHILE \ b \ DO \ c \quad \{P \land \neg b\}
\end{align*}
\]

In the While-rule, \( P \) is called an \textit{invariant} because it is preserved across executions of the loop body.
The *consequence* rule

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

\[ P' \rightarrow P \quad \{ P \} \ c \ \{ Q \} \quad Q \rightarrow Q' \]

\[ \{ P' \} \ c \ \{ Q' \} \]

*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.

\[
P \rightarrow Q[a/x]
\]

\[
\{P\} \quad x := a \quad \{Q\}
\]

\[
\{P \land b\} \quad c \quad \{P\} \quad P \land \neg b \rightarrow Q
\]

\[
\{P\} \quad \text{WHILE } b \ \text{DO } c \quad \{Q\}
\]
Example

\{ x = i \}

\texttt{y := 0;}

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

\{ y = \text{sum } i \}
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.
Partial Correctness

Introduction

The Syntactic Approach

The Semantic Approach

Soundness and Completeness
Assertions are predicates on states

\[ \text{assn} = \text{state} \Rightarrow \text{bool} \]

Alternative view: \textit{sets of states}

Semantic approach simplifies meta-theory, our main objective.
Validity

\[ \models \{ P \} \ c \ \{ Q \} \]

\[ \iff \]

\[ \forall \ s \ t. \ P \ s \land (c, s) \Rightarrow t \rightarrow Q \ t \]

“\{ P \} \ c \ \{ Q \} \ is \ valid”

In contrast:

\[ \vdash \{ P \} \ c \ \{ Q \} \]

“\{ P \} \ c \ \{ Q \} \ is \ provable/derivable”
Provability

\[ \vdash \{ P \} \text{ SKIP } \{ P \} \]

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

where \( s[a/x] \equiv s(x := \text{aval } a \ s) \)

Example: \( \{ x+5 = 5 \} \ x ::= x+5 \ \{ x = 5 \} \) in semantic terms:

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

where \( P = (\lambda s. \ (\lambda t. \ t \ x = 5)(s[\text{Plus} \ (V \ x) \ (N \ 5)/x])) \)

\[ = (\lambda s. \ (\lambda t. \ t \ x = 5)(s(x := s \ x + 5))) \]

\[ = (\lambda s. \ s \ x + 5 = 5) \]
\[
\begin{align*}
\vdash \{ P \} \ c_1 \ \{ Q \} & \quad \vdash \{ Q \} \ c_2 \ \{ R \} \\
\vdash \{ P \} \ c_1 \&; \ c_2 \ \{ R \} \\
\vdash \{ \lambda s. \ P \ s \ & \land \ bval \ b \ s \} \ c_1 \ \{ Q \} \\
\vdash \{ \lambda s. \ P \ s \ & \land \neg \ bval \ b \ s \} \ c_2 \ \{ Q \} \\
\vdash \{ P \} \ \text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \{ Q \} \\
\vdash \{ \lambda s. \ P \ s \ & \land \ bval \ b \ s \} \ c \ \{ P \} \\
\vdash \{ P \} \ \text{WHILE} \ b \ \text{DO} \ c \ \{ \lambda s. \ P \ s \ & \land \neg \ bval \ b \ s \} 
\end{align*}
\]
∀ s. P' s ⟷ P s
\vdash \{P\} c \{Q\}
∀ s. Q s ⟷ Q' s
\vdash \{P'\} c \{Q'\}
12 Partial Correctness

Introduction
The Syntactic Approach
The Semantic Approach

Soundness and Completeness
Soundness

Everything that is provable is valid:

\[ \vdash \{P\} \ c \ \{Q\} \implies \models \{P\} \ c \ \{Q\} \]

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 \( Q \):

\[
wp c Q = (\lambda s. \forall t. (c, s) \Rightarrow t \rightarrow Q t)
\]

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 \( \text{wp} \)

\[
\text{wp } \text{SKIP } Q = Q
\]

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

\[
\text{wp } (c_1 ;; c_2) \ Q = \text{wp } c_1 \ (\text{wp } c_2 \ Q)
\]

\[
\text{wp } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q = \ (\lambda s. \ \text{if } b\text{val } b \ s \ \text{then } \text{wp } c_1 \ Q \ s \ \text{else } \text{wp } c_2 \ Q \ s)
\]

\[
\neg \ b\text{val } b \ s \implies \text{wp } (\text{WHILE } b \ \text{DO } c) \ Q \ s = Q \ s
\]

\[
\text{bval } b \ s \implies \\
\text{wp } (\text{WHILE } b \ \text{DO } c) \ Q \ s = \\
\text{wp } (c ;; \ \text{WHILE } b \ \text{DO } c) \ Q \ s
\]
Completeness

\[ \models \{ P \} \ c \ \{ Q \} \implies \vdash \{ P \} \ c \ \{ Q \} \]

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

**Lemma** \[ \vdash \{ wp \ c \ Q \} \ c \ \{ Q \} \]

**Proof** by induction on \( c \), for arbitrary \( Q \).

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

**Fact** \[ \models \{ P \} \ c \ \{ Q \} \iff (\forall s. \ P \ s \implies wp \ c \ Q \ s) \]
Follows directly from defs of \( \models \) and \( wp \).
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 $\vdash$. 
12 Partial Correctness

13 Verification Conditions

14 Total Correctness
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: \( \text{pre} :: \text{com} \Rightarrow \text{assn} \Rightarrow \text{assn} \)
   - verification conditions: \( \text{vc} :: \text{com} \Rightarrow \text{assn} \Rightarrow \text{bool} \)

3. Soundness: \( \text{vc} \ c \ Q \implies \vdash \{ \ ? \} \ c \ \{ Q \} \)

4. Completeness: if \( \vdash \{ P \} \ c \ \{ Q \} \) then \( c \) can be annotated (becoming \( C \)) such that \( \text{vc} \ C \ Q \).

The details are a bit different ...
Annotated commands

Like commands, except for *While*:

**datatype** \( acom \) = \( \text{Skip} \)
| \( \text{Assign vname aexp} \)
| \( \text{Seq acom acom} \)
| \( \text{If bexp acom acom} \)
| \( \text{While assn bexp acom} \)

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

\( \{ I \} \ \text{WHILE} \ b \ \text{DO} \ c \)
Weakest precondition

\[ \text{pre} :: a\text{com} \Rightarrow \text{assn} \Rightarrow \text{assn} \]

\[ \text{pre \ SKIP} \ Q = Q \]

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

\[ \text{pre \ (C_1;; C_2)} \ Q = \text{pre \ C_1} \ (\text{pre \ C_2} \ Q) \]

\[ \text{pre \ (IF \ b \ THEN \ C_1 \ ELSE \ C_2)} \ Q = \]
\[ (\lambda s. \ \text{if \ bval \ b \ s \ then \ pre \ C_1 \ Q \ s \ else \ pre \ C_2 \ Q \ s}) \]

\[ \text{pre \ (\{ I \} \ WHILE \ b \ DO \ C)} \ Q = I \]
Warning

In the presence of loops, $\textit{pre } C$ may not be the weakest precondition but may be anything!
Verification condition

\(vc :: acom \Rightarrow assn \Rightarrow bool\)

\(vc \ SKIP \ Q = True\)

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

\(vc (C_1 ;; C_2) \ Q = (vc C_1 (pre C_2 Q) \land vc C_2 Q)\)

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

\(vc (\{I\} WHILE b DO C) \ Q = (\forall s. (I s \land bval b s \rightarrow pre C I s) \land (I s \land \neg bval b s \rightarrow Q s)) \land vc C I)\)
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 \(\textit{com}\), functions \(\textit{pre}\) and \(\textit{vc}\) operate on \(\textit{acom}\). Therefore we define

\[
\text{strip} :: \textit{acom} \Rightarrow \textit{com}
\]

\[
\text{strip} \ \text{SKIP} = \text{SKIP}
\]

\[
\text{strip} \ (x ::= a) = x ::= a
\]

\[
\text{strip} \ (C_1 ;; C_2) = \text{strip} \ C_1 ;; \text{strip} \ C_2
\]

\[
\text{strip} \ (\text{IF} \ b \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) = \text{IF} \ b \ \text{THEN} \ \text{strip} \ C_1 \ \text{ELSE} \ \text{strip} \ C_2
\]

\[
\text{strip} \ (\{ I \} \ \text{WHILE} \ b \ \text{DO} \ C) = \text{WHILE} \ b \ \text{DO} \ \text{strip} \ C
\]
Soundness of $\text{vc} \ & \ \text{pre}$ w.r.t. $\vdash$

$$\text{vc} \ C \ Q \iff \vdash \left\{\text{pre} \ C \ Q\right\} \ strip \ C \ \left\{Q\right\}$$

Proof by induction on $C$, for arbitrary $Q$.

Corollary:

$$\left[\text{vc} \ C \ Q; \ \forall \ s. \ P \ s \rightarrow \text{pre} \ C \ Q \ s\right]$$
$$\implies \vdash \left\{P\right\} \ strip \ C \ \left\{Q\right\}$$

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

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

But is premise provable if $\vdash \left\{P\right\} \ c \ \left\{Q\right\}$ is?
\[ \text{vc } C \; Q; \; \forall s. \; P \; s \rightarrow \text{pre } C \; Q \; s \] 
\[ \rightarrow \; \vdash \{ P \} \; \text{strip } C \; \{ Q \} \]

Why could premise not be provable although conclusion is?

- Some annotation in $C$ is not invariant.
- $\text{vc}$ or $\text{pre}$ 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. $\vdash$

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

$\exists C. \ strip \ C = c \land \ vc \ C \ Q \land (\forall s. \ P \ s \implies pre \ C \ Q \ s)$

Proof by rule induction. Needs two monotonicity lemmas:

$[\forall s. \ P \ s \implies P' \ s; \ pre \ C \ P \ s] \implies pre \ C \ P' \ s$

$[\forall s. \ P \ s \implies P' \ s; \ vc \ C \ P] \implies vc \ C \ P'$
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:

\[
(\models_t \{ P \} c \{ Q \}) = \quad (\forall s. \ P \ s \rightarrow (\exists t. \ (c, \ s) \Rightarrow t \land Q \ t))
\]

Assumes that semantics is deterministic!

Exercise: Reformulate for nondeterministic language
⊢: A proof system for total correctness

Only need to change the \texttt{WHILE} rule.

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

\[
\forall n. \vdash_t \{ \lambda s. P s \land \text{bval } b s \land n = f s \} \quad c \quad \{ \lambda s. P s \land f s < n \}
\]

\[
\vdash_t \{ P \} \quad \text{WHILE } b \quad \text{DO } c \quad \{ \lambda s. P s \land \neg \text{bval } b s \}
\]
**WHILE** rule can be generalized from a function to a relation:

\[ \bigwedge n. \vdash_t \{ \lambda s. P s \land bval b s \land T s n \} \quad c \quad \{ \lambda s. P s \land (\exists n' < n. T s n') \} \]

\[ \vdash_t \{ \lambda s. P s \land (\exists n. T s n) \} \quad \text{WHILE} \quad b \quad \text{DO} \quad c \quad \{ \lambda s. P s \land \neg bval b s \} \]
Hoare_Total.thy

Example
Soundness

\[ \vdash_t \{ P \} \ c \ \{ Q \} \implies \models_t \{ P \} \ c \ \{ Q \} \]

Proof by induction, with a nested induction on \( n \) in the While-case.
Completeness

\[ \models_t \{ P \} \ c \ \{ Q \} \implies \vdash_t \{ P \} \ c \ \{ Q \} \]

Follows easily from

\[ \vdash_t \{ \text{wp}_t \ c \ Q \} \ c \ \{ Q \} \]

where

\[ \text{wp}_t \ c \ Q = (\lambda s. \exists t. (c, s) \Rightarrow t \land Q \ t) . \]
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

\[
\begin{align*}
\neg \ bval \ b \ s & \quad bval \ b \ s \quad (c, s) \Rightarrow s' \quad T \ s' \ n \\
T \ s \ 0 & \quad T \ s \ (n + 1)
\end{align*}
\]

$T \ s \ n$ means that WHILE $b$ DO $c$ started in state $s$ needs $n$ iterations to terminate.
Chapter 13

Abstract Interpretation
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination

Analysis of Boolean Expressions

Widening and Narrowing
• 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:
  \[
  a[i] := 1; \ a[j] := 2; \ x := a[i] \quad \rightarrow \quad a[i] := 1; \ a[j] := 2; \ x := 1
  \]
  if \( i \neq j \)
- ...
Applications: Debugging/Verification

Detect presence or absence of certain runtime exceptions/errors:

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

- Null pointer analysis
  - ...

- ...
Precision

A consequence of Rice’s theorem:

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

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

If an analysis says that some value

- 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:

\[
x := 0 \{ \ldots \}; \ y := 0 \{ \ldots \}
\]
Annotated WHILE

**View**

\[
\{ Inv \} \quad \text{WHILE} \quad b \quad \text{DO} \quad \{ P \} \quad c \\
\{ Q \}
\]

as a *control flow graph* with annotated nodes:
The starting point:
Collecting Semantics

Collects all possible states for each program point:

\[
\begin{align*}
\text{x} & := 0 \{ <x := 0> \} ; \\
& \{ <x := 0>, <x := 2>, <x := 4> \} \\
\text{WHILE} \ x < 3 \\
\text{DO} \ & \{ <x := 0>, <x := 2> \} \\
& \quad x := x + 2 \{ <x := 2>, <x := 4> \} \\
& \quad \{ <x := 4> \}
\end{align*}
\]
Infinite sets of states:

\{ \ldots, \langle x := -1 \rangle, \langle x := 0 \rangle, \langle x := 1 \rangle, \ldots \} \quad \text{WHILE } x < 3

DO \{ \ldots, \langle x := 1 \rangle, \langle x := 2 \rangle \}

\quad x := x+2 \{ \ldots, \langle x := 3 \rangle, \langle x := 4 \rangle \}

\{ \langle x := 3 \rangle, \langle x := 4 \rangle, \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

\[(vname \Rightarrow val) \text{ set} \sim vname \Rightarrow val \text{ set}\]

\[
x := 0 \{ <x := \{0\}> \} ;
\{ <x := \{0,2,4\}> \}
\]

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

\[
< x:=0, \ y:=1 > \enspace \text{and} \enspace < x:=1, \ y:=0 >.
\]
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:

```plaintext
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.
\[ x := 0 \]

\[ \text{WHILE } x < 3 \]
\[ \text{DO} \]
\[ x := x+2 \]
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination

Analysis of Boolean Expressions

Widening and Narrowing
Concrete syntax

\[ 'a \ acom \ ::= \ \text{SKIP} \ \{ 'a \} \ \mid \ \text{string} ::= \ aexp \ \{ 'a \} \ \mid \ 'a \ acom \ ;; \ 'a \ acom \]

\[ \mid \ 'a \ acom \ ;; \ 'a \ acom \]

\[ \mid \ \text{IF} \ bexp \ \text{THEN} \ \{ 'a \} \ 'a \ acom \]

\[ \text{ELSE} \ \{ 'a \} \ 'a \ acom \]

\[ \{ 'a \} \]

\[ \mid \ \{ 'a \} \]

\[ \text{WHILE} \ bexp \ \text{DO} \ \{ 'a \} \ 'a \ acom \]

\[ \{ 'a \} \]

\[ 'a: \text{ type of annotations} \]

Example: \[ "x" ::= \ N \ 1 \ \{ 9 \} \ ;; \ \text{SKIP} \ \{ 6 \} :: \ \text{nat} \ acom \]
Abstract syntax

datatype

'\texttt{a} acom =

\texttt{SKIP} 'a

| Assign string aexp 'a
| \texttt{Seq} ('a acom) ('a acom)
| If bexp 'a ('a acom) 'a ('a acom) 'a
| While 'a bexp 'a ('a acom) 'a
Auxiliary functions: $\text{strip}$

Strips all annotations from an annotated command

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

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

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

$\text{strip} (C_1 ;; C_2) = \text{strip } C_1 ;; \text{strip } C_2$

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

$= \text{IF } b \text{ THEN } \text{strip } C_1 \text{ ELSE } \text{strip } C_2$

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

$= \text{WHILE } b \text{ DO } \text{strip } C$

We call $C$ and $C'$ $\text{strip-equal}$ iff $\text{strip } C = \text{strip } C'$.
Auxiliary functions: \textit{annos}

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

\texttt{annos} :: 'a acom ⇒ 'a list

\texttt{annos} (\texttt{SKIP} \{P\}) = \[P\]

\texttt{annos} (x ::= e \{P\}) = \[P\]

\texttt{annos} (C_1;; C_2) = \texttt{annos} C_1 @ \texttt{annos} C_2

\texttt{annos} (IF b THEN \{P_1\} C_1 ELSE \{P_2\} C_2 \{Q\}) = P_1 \# \texttt{annos} C_1 @ P_2 \# \texttt{annos} C_2 @ [Q]

\texttt{annos} (\{I\} WHILE b DO \{P\} C \{Q\}) = I \# P \# \texttt{annos} C @ [Q]
Auxiliary functions: \( \textit{anno} \)

\[
\text{anno :: } 'a \text{ acom } \Rightarrow \text{ nat } \Rightarrow \ 'a \\
\text{anno } C \ p = \text{ annos } C \ ! \ p
\]

The \( p \)-th annotation (starting from 0)
Auxiliary functions:  \textit{post}

\[
\begin{align*}
\text{post} & : \ 'a \ acom \Rightarrow \ 'a \\
\text{post } C & = \text{last } (\text{annos } C)
\end{align*}
\]

The rightmost/last/post annotation
Auxiliary functions: \textit{map\_acom}

\[\text{map\_acom} :: (\,a \Rightarrow \,b) \Rightarrow \,a \text{ acom} \Rightarrow \,b \text{ acom}\]

\text{map\_acom } f \ C \text{ applies } f \text{ to all annotations in } C
Introduction
Annotated Commands
Collecting Semantics
Abstract Interpretation: Orderings
A Generic Abstract Interpreter
Executable Abstract State
Termination
Analysis of Boolean Expressions
Widening and Narrowing
Annotate commands with the set of states that can occur at each annotation point.

The annotations are generated iteratively:

\[ \text{step} :: \text{state set} \Rightarrow \text{state set acom} \Rightarrow \text{state set acom} \]

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

\[ \text{start states} \]

flowing into the command
\text{step} \quad 

\text{step } S \left( SKIP \{ - \} \right) = SKIP \{ S \}

\text{step } S \left( x ::= e \{ - \} \right) = 
\quad x ::= e \left\{ \{ s(x := \text{aval} e s) \mid s. \ s \in S \} \right\}

\text{step } S \left( C_1 ;; C_2 \right) = \text{step } S \ C_1 ;; \text{step } \left( \text{post } C_1 \right) \ C_2

\text{step } S \left( IF \ b \ \text{THEN} \ \{ P_1 \} \ C_1 \ \text{ELSE} \ \{ P_2 \} \ C_2 \ \{ - \} \right) = 
\quad IF \ b \ \text{THEN} \ \{ \{ s \in S. \ \text{bval} b s \} \} \ \text{step } P_1 \ C_1
\quad \text{ELSE} \ \{ \{ s \in S. \ \neg \text{bval} b s \} \} \ \text{step } P_2 \ C_2
\quad \{ \text{post } C_1 \cup \text{post } C_2 \}
\[
\text{step } S (\{I\} \text{ WHILE } b \text{ DO } \{P\} \text{ C } \{\_\}) = \\
\{S \cup \text{post C}\} \\
\text{WHILE } b \\
\text{DO } \{\{s \in I. \ bval \ b \ s\}\} \\
\text{step } P \text{ C} \\
\{\{s \in I. \neg bval \ b \ s\}\}
\]
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:

$$\text{step } S \ C = C$$
Collecting_Examples.thy
Abstract example

Let \( C = \{ I \} \)

\[
\text{WHILE } b \\
\text{DO } \{ P \} C_0 \\
\{ Q \}
\]

\( \text{step } S C = C \) means

\[
I = S \cup \text{post } C_0 \\
P = \{ s \in I. \ bval b s \} \\
C_0 = \text{step } P C_0 \\
Q = \{ s \in I. \lnot \ bval b s \}
\]

Fixpoint = solution of equation system

Iteration is just one way of solving equations
Why *least* fixpoint?

\[
\{ I \} \\
\text{WHILE true} \\
\text{DO } \{ I \} \text{ SKIP } \{ I \} \\
\{ \{ \} \} \\
\]

Is fixpoint of *step* \{ \} for every \( I \)

But the “reachable” fixpoint is \( I = \{ \} \)
Does \textit{step} always have a least fixpoint?
A type ‘a is a **partial order** if

- there is a predicate \( \leq :: \text{‘}a \Rightarrow \text{‘}a \Rightarrow \text{bool} \)
- that is **reflexive** \( (x \leq x) \),
- **transitive** \( ([x \leq y;\ y \leq z] \implies x \leq z) \) and
- **antisymmetric** \( ([x \leq y;\ y \leq x] \implies x = y) \)
Complete lattice

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

- \( \forall s \in S. \ l \leq s \)
- If \( \forall s \in S. \ l' \leq s \) then \( l' \leq l \)

The greatest lower bound (\textit{infimum}) of \( S \) is often denoted by \( \bigwedge S \).

\textbf{Fact} Type \ 'a set \ is a complete lattice where \( \leq = \subseteq \) and \( \bigwedge = \bigcap \)
Lemma In a complete lattice, every set \( S \) of elements also has a *least upper bound* \((\text{supremum})\) \( \bigcup S \):

- \( \forall s \in S. \ s \leq \bigcup S \)
- If \( \forall s \in S. \ s \leq u \) then \( \bigcup S \leq u \)

The least upper bound is the greatest lower bound of all upper bounds: \( \bigcup S = \bigcap \{u. \ \forall s \in S. \ s \leq u\} \).

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 $x \leq y \implies f(x) \leq f(y)$.

**Theorem** (Knaster-Tarski) Every monotone function on a complete lattice has the least (pre-)fixpoint

$$\bigcap \{p. f(p) \leq p\}.$$  

**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:

\[ C_1 \leq C_2 \iff \text{strip } C_1 = \text{strip } C_2 \land \\
(\forall p < \text{length } (\text{annos } C_1). \ \text{anno } C_1 p \leq \text{anno } C_2 p) \]

Lemma If ′a is a partial order, so is ′a acom.
Example:

\[
x ::= N \ 0 \ \{\{a\}\} \leq x ::= N \ 0 \ \{\{a, \ b\}\} \quad \iff \quad True
\]

\[
x ::= N \ 0 \ \{\{a\}\} \leq x ::= N \ 0 \ \{\}\quad \iff \quad False
\]

\[
x ::= N \ 0 \ \{S\} \leq x ::= N \ 1 \ \{S\} \quad \iff \quad False
\]
The collecting semantics needs to order \textit{state set acom}.

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

Does \textit{state set acom} also form a complete lattice?

Almost . . .
What is the infimum of \( \text{SKIP} \{ S \} \) and \( \text{SKIP} \{ T \} \)?

\[ \text{SKIP} \{ S \cap T \} \]

What is the infimum of \( \text{SKIP} \{ S \} \) and \( x ::= \mathbb{N} \ 0 \{ T \} \)?

Only \textit{strip}-equal commands have an infimum.
It turns out:

- if 'a is a complete lattice,
- then for each \( c :: com \)
- the set \( \{ C :: 'a acom. \text{strip } C = c \} \)
  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 $\bigwedge M \in L$.

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

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

$$\cap \ \{ \SKIP \{ A \}, \SKIP \{ B \}, \ldots \}$$

$$= \SKIP \{ \cap \ \{ A, B, \ldots \} \}$$

Formally . . .
Auxiliary function: \textit{annotate}

\[\text{annotate} :: (\text{nat} \Rightarrow \textquote{a}) \Rightarrow \text{com} \Rightarrow \textquote{a} \text{acom}\]

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

\[\textit{anno} \ (\text{annotate} \ f \ c) \ p = f \ p\]
Lemma Let \( 'a \) be a complete lattice and \( c :: \text{com} \).

Then \( L = \{ C :: 'a \text{ acom. strip } C = c \} \)
is a complete lattice where the infimum of \( M \subseteq L \) is

\[
\text{annotate} \ (\lambda p. \bigwedge \{ \text{anno } C \ p \ \mid C. \ C \in M \}) \ c
\]

Proof straightforward (pointwise).
The Collecting Semantics

The underlying complete lattice is now \textit{state set}.

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

\textbf{Lemma} \( \text{step } S \in L \rightarrow L \) and is monotone.

Therefore Knaster-Tarski is applicable and we define

\[
CS :: \text{com} \Rightarrow \text{state set acom}
\]

\[
CS \ c = \lfp \ c \ (\text{step UNIV})
\]

[\( \lfp \) 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** \((c, s) \Rightarrow t \iff t \in \text{post} (CS c)\)

Follows directly from

\[
\llbracket (c, s) \Rightarrow t; s \in S \rrbracket \iff t \in \text{post}(\text{lfp } c (\text{step } S))
\]
Proof of

\[ (c, s) \Rightarrow t; \ s \in S \] \implies t \in \text{post}(\text{lfp } c (\text{step } S))

uses

\[ \text{post}(\text{lfp } c f) = \bigcap \{ \text{post } C \mid C. \ \text{strip } C = c \land f\ C \leq C\} \]

and

\[ [(c, s) \Rightarrow t; \ \text{strip } C = c; \ s \in S; \ \text{step } S \ C \leq C] \implies t \in \text{post } C \]

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

\[ t \in \text{post}(\text{lfp } c \ (\text{step } S)) \implies \exists s \in S. \ (c,s) \Rightarrow t \]

is also true but is not proved in this course.
Approximating the Collecting semantics

A conceptual step:

$$(vname \Rightarrow val) \; set \; \leadsto \; vname \Rightarrow val \; set$$

A domain-specific step:

$$val \; set \; \leadsto \; 'av$$

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

Abstract values:

**datatype** \( parity = \text{Even} \mid \text{Odd} \mid \text{Either} \)

\[
egin{array}{c}
\text{Either} \\
\text{Even} \\
\text{Odd} \\
\mathbb{Z} \\
2\mathbb{Z} \\
2\mathbb{Z} + 1 \\
\end{array}
\]

concretization function \( \gamma_{parity} \)
A concretisation function $\gamma$
maps an abstract value to a set of concrete values

Bigger abstract values represent more concrete values
Example: parity

Fact Type \textit{parity} is a partial order.
A partial order \( 'a \) has a \textit{top element} \( \top \) if \( 'a \) if

\[
a \leq \top
\]
A type `'a` is a **semilattice** if

- it is a partial order and
- there is a least upper bound operation

\[
\sqcup :: \:'a \Rightarrow \:'a \Rightarrow \:'a
\]

\[
x \leq x \sqcup y \quad y \leq x \sqcup y
\]

\[
[x \leq z; \ y \leq z] \implies x \sqcup y \leq z
\]

Application: abstract \( \sqcup \), join two computation paths

We often call \( \sqcup \) the **join** operation.
Fact If $\mathcal{A}$ 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 $u_2 \leq u_1$. 

$\leq$ uniquely determines $\sqcup$
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

```plaintext
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 \( \tau \) belongs to class \( C \)

Example: \( parity :: \text{semilattice_sup} \)
Orderings and instances
From abstract values to abstract states

Need to abstract collecting semantics:

\[ \text{state set} \]

- 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 $\gamma$ to
'av st option

Lemma
If $\mathcal{V}$ :: semilattice_sup_top
then $\mathcal{V} \Rightarrow \mathcal{A}$ :: 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
$\gamma_{\text{fun}} :: (\mathcal{V} \Rightarrow \mathcal{C} \ \text{set}) \Rightarrow (\mathcal{V} \Rightarrow \mathcal{A}) \Rightarrow (\mathcal{V} \Rightarrow \mathcal{C}) \ \text{set}$
where $\gamma_{\text{fun}} \gamma \ F = \{ f. \forall x. f \ x \in \gamma (F \ x) \}$

Lemma
If $\gamma$ is monotone then $\gamma_{\text{fun}} \gamma$ is monotone.
Lemma  If 'a :: semilattice_sup_top
    then 'a option :: semilattice_sup_top

Proof
(Some x ≤ Some y) = (x ≤ y)
(None ≤ _) = True
(Some _ ≤ None) = False

Some x ⊔ Some y = Some (x ⊔ y)
None ⊔ y = y
x ⊔ None = x
⊤ = Some ⊤

Corollary  If 'a :: semilattice_sup_top
    then 'a st option :: semilattice_sup_top
fun $\gamma_{\text{option}} :: (\forall a \Rightarrow \forall c \, \text{set}) \Rightarrow \forall a \, \text{option} \Rightarrow \forall c \, \text{set}$
where
$\gamma_{\text{option}} \, \gamma \, \text{None} = \{\}$
$\gamma_{\text{option}} \, \gamma \, (\text{Some} \ a) = \gamma \ a$

**Lemma** If $\gamma$ is monotone then $\gamma_{\text{option}} \, \gamma$ is monotone.
Remember:

**Lemma** If \('a :: \text{order}\) then \('a \text{acom} :: \text{order}\).

Partial order is enough, semilattice not needed.

Lifting \(\gamma :: \ 'a \Rightarrow \ 'c\) to \('a \text{acom} \Rightarrow \ 'c \text{acom}\) is easy:

\[
\text{map}_\text{acom}
\]

**Lemma**

If \(\gamma\) is monotone then \(\text{map}_\text{acom} \gamma\) is monotone.
15 Introduction
16 Annotated Commands
17 Collecting Semantics
18 Abstract Interpretation: Orderings
19 A Generic Abstract Interpreter
20 Executable Abstract State
21 Termination
22 Analysis of Boolean Expressions
23 Widening and Narrowing
• 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 \( val = \texttt{int} \).

• Result/Output: abstract interpreter that approximates the collecting semantics by computing on abstract values.

• Realization in Isabelle as a \textit{locale}
Abstract values: type  \( 'av :: \text{semilattice\_sup\_top} \)

Concretization function:  \( \gamma :: 'av \Rightarrow \text{val set} \)

Assumptions:  
\[
\begin{align*}
  a \leq b & \implies \gamma a \subseteq \gamma b \\
  \gamma \top & = \text{UNIV}
\end{align*}
\]
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:
\[ i \in \gamma (num' i) \]
\[ [i_1 \in \gamma a_1; i_2 \in \gamma a_2] \implies i_1 + i_2 \in \gamma (plus' a_1 a_2) \]

The \( n \in \gamma a \) relationship is maintained
Lifted concretization functions

\[ \gamma_s :: \text{'av st} \Rightarrow \text{state set} \]
\[ \gamma_s = \gamma_{\text{fun}} \gamma \]

\[ \gamma_o :: \text{'av st option} \Rightarrow \text{state set} \]
\[ \gamma_o = \gamma_{\text{option}} \gamma_s \]

\[ \gamma_c :: \text{'a st option acom} \Rightarrow \text{state set acom} \]
\[ \gamma_c = \text{map_acom} \gamma_o \]

All of them are monotone.
Abstract interpretation of $aexp$

\begin{align*}
\textbf{fun} \; \textit{aval}' :: \; aexp & \Rightarrow \; 'av \; st \Rightarrow \; 'av \\
\textit{aval}' \; (N \; n) \; S & = \; \textit{num}' \; n \\
\textit{aval}' \; (V \; x) \; S & = \; S \; x \\
\textit{aval}' \; (\textit{Plus} \; a_1 \; a_2) \; S & = \; \textit{plus}' \; (\textit{aval}' \; a_1 \; S) \; (\textit{aval}' \; a_2 \; S)
\end{align*}

Correctness of $\textit{aval}'$ wrt $\textit{aval}$:

\textbf{Lemma} $\; s \in \gamma \; S \rightleftarrows \; \textit{aval} \; a \; s \in \gamma \; (\textit{aval}' \; a \; S)$

\textbf{Proof} by induction on $a$

using the assumptions about the parameters.
Example instantiation with $\textit{parity}$

$\leq_{\sqcup}$ and $\gamma_{\textit{parity}}$: see earlier

\begin{align*}
\text{num}_{\textit{parity}} \ i &= (\text{if } i \mod 2 = 0 \text{ then Even else Odd}) \\
\text{plus}_{\textit{parity}} \ \text{Even} \ \text{Even} &= \text{Even} \\
\text{plus}_{\textit{parity}} \ \text{Odd} \ \text{Odd} &= \text{Even} \\
\text{plus}_{\textit{parity}} \ \text{Even} \ \text{Odd} &= \text{Odd} \\
\text{plus}_{\textit{parity}} \ \text{Odd} \ \text{Even} &= \text{Odd} \\
\text{plus}_{\textit{parity}} \ \text{Either} \ y &= \text{Either} \\
\text{plus}_{\textit{parity}} \ x \ \text{Either} &= \text{Either}
\end{align*}
Example instantiation with \textit{parity}

**Input:**
\begin{align*}
\gamma & \mapsto \gamma_{parity} \\
\text{num'} & \mapsto \text{num\_parity} \\
\text{plus'} & \mapsto \text{plus\_parity}
\end{align*}

**Must prove parameter assumptions**

**Output:**
\begin{align*}
\text{aval'} & \mapsto \text{aval\_parity}
\end{align*}

**Example** The value of

\[
\text{aval\_parity} \ (\text{Plus} \ (V \ "x") \ (V \ "x")) \\
((\lambda_. \ \text{Either})("x" := \text{Odd}))
\]

is \textit{Even}.
Abs_Int1_parity.thy

Locale interpretation
Abstract interpretation of $bexp$

For now, boolean expressions are not analysed.
Abstract interpretation of \textit{com}

Abstracting the collecting semantics

\[
\text{step} :: \quad \tau \Rightarrow \tau \text{ acom} \Rightarrow \tau \text{ acom}
\]

where \( \tau = \text{state set} \)

to

\[
\text{step'} :: \quad \tau \Rightarrow \tau \text{ acom} \Rightarrow \tau \text{ acom}
\]

where \( \tau = '\text{av st option} \)

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

\[
\text{Step} :: 'a \Rightarrow 'a \text{ acom} \Rightarrow 'a \text{ acom}
\]
Step :: 'a ⇒ 'a acom ⇒ 'a acom

Parameterized wrt
- type 'a with □
- the interpretation of assignments and tests:
  asem :: vname ⇒ aexp ⇒ 'a ⇒ 'a
  bsem :: bexp ⇒ 'a ⇒ 'a
Step \( a (SKIP \{\_\}) = SKIP \{a\} \)

Step \( a (x ::= e \{\_\}) = x ::= e \{a \text{sem } x \ e \ a\} \)

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

Step \( a \ (\text{IF } b \ \text{THEN} \ \{P_1\} \ C_1 \ \text{ELSE} \ \{P_2\} \ C_2 \ \{\_\}) = \)
IF \( b \ \text{THEN} \ \{b \text{sem } b \ a\} \ \text{Step } P_1 \ C_1 \) 
ELSE \( \{b \text{sem } (\text{Not } b) \ a\} \ \text{Step } P_2 \ C_2 \) 
\(\{post \ C_1 \sqcap post \ C_2\}\)

Step \( a \ (\{I\} \ \text{WHILE } b \ \text{DO} \ \{P\} \ C \ \{\_\}) = \)
\(\{a \sqcap post \ C\} \ \text{WHILE } b \ \text{DO} \ \{b \text{sem } b \ I\} \ \text{Step } P \ C \) 
\(\{b \text{sem } (\text{Not } b) \ I\}\)
Instantiating \textit{Step}

The truth: \textit{asem} and \textit{bsem} are (hidden) parameters of \textit{Step}: \textit{Step asem bsem ...}

\[
\text{step} = \\
\text{Step} \ (\lambda x\ e\ S. \ \{s(x := \text{aval} e \ s) \mid \text{s. } s \in S\}) \\
(\lambda b\ S. \ \{s \in S. \ bval b \ s\})
\]

\[
\text{step}' = \text{Step asem} \ (\lambda b\ S. \ S)
\]

where

\[
\text{asem } x\ e\ S = \\
(\text{case } S\ \text{of } None \Rightarrow None \\
\mid \text{Some } S \Rightarrow \text{Some } (S(x := \text{aval}' e \ S)))
\]
Example: iterating $step\_parity$

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

where

$$C = \ x ::= N \ 3 \ \{None\} ; \ 
\{None\} \ 
WHILE \ b \ DO \ \{None\} \ 
\ x ::= Plus (V x) (N \ 5) \ \{None\} \ 
\{None\}$$

$$S = \ Some \ (\lambda_. \ Either)$$

$$S_p = \ Some \ ((\lambda_. \ Either)(x ::= p))$$
Correctness of \( step' \) wrt \( step \)

The conretization of \( step' \) overapproximates \( step \):

**Corollary** \( step (\gamma_o S) (\gamma_c C) \leq \gamma_c (step' S C) \)

where \( S :: \) 'av st option
\( C :: \) 'av st option acom

**Lemma** \( \text{Step } f \; g \; (\gamma_o S) \; (\gamma_c C) \leq \gamma_c \; (\text{Step } f' \; g' \; S \; C) \)

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

From the HOL library:

\[
\text{while} \_ \text{option} ::
\]
\[
('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ option}
\]

such that

\[
\text{while} \_ \text{option} b f x =
\]
\[
(\text{if } b x \text{ then } \text{while} \_ \text{option} b f (f x) \text{ else } \text{Some } x)
\]

and  \text{while} \_ \text{option} b f x = \text{None}

if the recursion does not terminate.
Pre-fixpoint:

\[ pfp :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ option} \]

\[ pfp f = \text{whileOption} (\lambda x. \neg f \, x \leq x) \, f \]

Start iteration with least annotated command:

\[ \text{bot} \, c = \text{annotate} (\lambda p. \text{None}) \, c \]
The generic abstract interpreter

definition $AI :: \text{com} \Rightarrow 'av \text{st} \text{option} \text{acom} \text{option}$

where $AI \ c = \text{pfp} \ (\text{step'} \ \top) \ (\text{bot} \ c)$

Theorem $AI \ c = \text{Some} \ C \implies CS \ c \leq \gamma_c \ C$

Proof From the assumption: $\text{step'} \ \top \ C \leq C$

By monotonicity: $\gamma_c \ (\text{step'} \ \top \ C) \leq \gamma_c \ C$

By step/step': $\text{step} \ (\gamma_o \ \top) \ (\gamma_c \ C) \leq \gamma_c \ (\text{step'} \ \top \ C)$

Hence $\gamma_c \ C$ is a pfp of $\text{step} \ (\gamma_o \ \top) = \text{step} \ \text{UNIV}$

Because $CS$ is the least pfp of $\text{step} \ \text{UNIV}$: $CS \ c \leq \gamma_c \ C$
AI is not directly executable

because \( pfp \) compares \( f \ C \leq C \)
where \( C :: 'av \ st \ option \ acom \)
which compares functions \( vname \Rightarrow 'av \)
which is not computable: \( vname \) is infinite.
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination

Analysis of Boolean Expressions

Widening and Narrowing
Solution

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

An association list representation:

type_synonym 'a st_rep = (vname × 'a) list

From 'a st_rep back to vname ⇒ 'a:

fun fun_rep :: ('a::top) st_rep ⇒ (vname ⇒ 'a)
fun_rep ((x, a) # ps) = (fun_rep ps)(x := a)
fun_rep [] = (λx. ⊤)

Missing variables are mapped to ⊤

Example: fun_rep [("x", a), ("x", b)]
= ((λx. ⊤)("x" := b))("x" := a) = (λx. ⊤)("x" := a)
Comparing association lists

Compare them only on their finite “domains”:

\[
\text{less_eq_st_rep } ps_1 \ ps_2 = \\
(\forall \ x \in \text{set } \ (\text{map } \text{fst } \ ps_1)) \cup \text{set } \ (\text{map } \text{fst } \ ps_2). \\
\quad \text{fun_rep } ps_1 \ x \leq \text{fun_rep } ps_2 \ x
\]

Not a partial order because not antisymmetric!

Example: \[(["x", a), ("y", b)] \text{ and } (["y", b), ("x", a)]\]
Quotient type 'a st

Define \( \text{eq}_st \, ps_1 \, ps_2 = (\text{fun}_\text{rep} \, ps_1 = \text{fun}_\text{rep} \, ps_2) \)

Overwrite 'a st = \( \text{vname} \Rightarrow 'a \) by

\textbf{quotient\_type} 'a st = ('a::top) st\_rep / eq\_st

Elements of 'a st:

\textit{equivalence classes} \( [ps]_{eq\_st} = \{ps'. \, eq\_st \, ps \, ps'\} \)

Abbreviate \( [ps]_{eq\_st} \) 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:
\[ \text{fun} \ (St \ ps) = \text{fun}_\text{rep} \ ps \]

Updating an abstract state:
\[ \text{update} \ (St \ ps) \ x \ a = St \ ((x, a) \# \ ps) \]
Turning 'a st into a semilattice

\[(St \, ps_1 \leq St \, ps_2) = \text{less_eq_st_rep} \, ps_1 \, ps_2\]

\[St \, ps_1 \sqcup St \, ps_2 = St(\text{map2_st_rep} \, (op \sqcup) \, ps_1 \, ps_2)\]

```plaintext
fun map2_st_rep ::
(\'a::top \Rightarrow \'a \Rightarrow \'a) \Rightarrow \'a st_rep \Rightarrow \'a st_rep \Rightarrow \'a st_rep
map2_st_rep \, f \, ((x, y) \# \, ps_1) \, ps_2 =
let \, y_2 = fun_rep \, ps_2 \, x
in \, (x, f \, y \, y_2) \# \, map2_st_rep \, f \, ps_1 \, ps_2
map2_st_rep \, f \, [] \, ps_2 = \text{map} \, (\lambda(x, \, y). \, (x, f \top \, y)) \, ps_2
```
Modified abstract interpreter

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

$S \ x \ \leadsto \ \text{fun} \ S \ x$

$S(x := a) \ \leadsto \ \text{update} \ S \ x \ a$

Now $AI$ is executable!
Abs_Int1_parity.thy
Abs_Int1_const.thy

Examples
15 Introduction
16 Annotated Commands
17 Collecting Semantics
18 Abstract Interpretation: Orderings
19 A Generic Abstract Interpreter
20 Executable Abstract State
21 Termination
22 Analysis of Boolean Expressions
23 Widening and Narrowing
Beyond partial correctness

- $AI$ may compute any pfp
- $AI$ may not terminate

The solution: Monotonicity

→

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:

\[
\begin{bmatrix} a_1 \leq b_1; \ a_2 \leq b_2 \end{bmatrix} \implies \text{plus}' \ a_1 \ a_2 \leq \text{plus}' \ b_1 \ b_2
\]

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

\[
S_1 \leq S_2 \implies \text{aval}' \ e \ S_1 \leq \text{aval}' \ e \ S_2
\]

and therefore *step'* is also monotone:

\[
\begin{bmatrix} S_1 \leq S_2; \ C_1 \leq C_2 \end{bmatrix} \implies \text{step}' \ S_1 \ C_1 \leq \text{step}' \ S_2 \ C_2
\]
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 $\leq$ be a partial order on a set $L$ with least element $\bot \in L$: $x \in L \implies \bot \leq x$.
Let $f \in L \rightarrow L$ be a monotone function.
If while_option (\(\lambda x. \neg f x \leq x\)) $f \bot = \text{Some } p$ then $p$ is the least pre-fixpoint of $f$ on $L$.
That is, if $f q \leq q$ for some $q \in L$, then $p \leq q$.

Proof Clearly $f p \leq p$.
Given any pre-fixpoint $q \in L$, property
\[
P x = (x \in L \land x \leq q)
\]
is an invariant of the while loop:
\[
P \bot \text{ holds and } P x \text{ implies } f x \leq f q \leq q
\]
Hence upon termination $P p$ must hold and thus $p \leq q$. 
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
Because $step'$ is monotone, starting from $bot c$ generates an ascending $<\text{ chain of annotated commands.}$ We exhibit a measure function $m_c$ that decreases with every loop iteration:

$$C_1 < C_2 \implies m_c C_2 < m_c C_1$$

Modulo some details . . .
The measure function $m_c$ is constructed from a measure function $m$ on $\text{'av}$ in several steps.

Parameters: $m :: \text{'av} \Rightarrow \text{nat}$

$$h :: \text{nat}$$

Assumptions: $m x \leq h$

$$x < y \implies m y < m x$$

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

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

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

\( m_s :: \text{'av st} \Rightarrow \text{vname set} \Rightarrow \text{nat} \)
\( m_s \ S \ X = (\sum x \in X. \ m \ (S \ x)) \)
All measure functions are bounded:

\[
\text{finite } X \implies m_s S X \leq h \times \text{card } X
\]

\[
\text{finite } X \implies m_o \text{ opt } X \leq h \times \text{card } X + 1
\]

\[
m_c C \leq \text{length (annos } C) \times (h \times \text{card (vars } C) + 1)
\]

Therefore \( AI c \) requires at most \( p \times (h \times n + 1) \) 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:

\[ \text{finite } X \implies S_1 < S_2 \implies m_s S_2 X < m_s S_1 X \]

because \( S_1 < S_2 \iff S_1 \leq S_2 \land (\exists x. S_1 x < S_2 x) \)

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:

\[
[ \text{finite } X; \ S_1 = S_2 \text{ on } - X; \ S_1 < S_2 ] \\
\implies m_s S_2 X < m_s S_1 X
\]

\[
[ \text{finite } X; \ \text{top}\_\text{on}_o o_1 ( - X); \ \text{top}\_\text{on}_o o_2 ( - X); \ o_1 < o_2 ] \\
\implies m_o o_2 X < m_o o_1 X
\]

\[
[ \text{top}\_\text{on}_c C_1 ( - \text{vars } C_1); \ \text{top}\_\text{on}_c C_2 ( - \text{vars } C_2); \ C_1 < C_2 ] \\
\implies m_c C_2 < m_c C_1
\]
Now we can prove termination

\[ \exists C. AI \, c = \text{Some } C \]

because \textit{step}' leaves \textit{top_on}s invariant:

\[
\begin{align*}
\text{top_on}_c \quad & C \ (\neg \text{ vars } \ C) \quad \Rightarrow \\
\text{top_on}_c \quad (\text{step}' \ \sqcap \ C) \ (\neg \text{ vars } \ C)
\end{align*}
\]
Warning: \textit{step'} is very inefficient. 
It is applied to every subcommand in every step. 
Thus the actual complexity of \textit{AI} is $O(p^2 \ast n \ast h)$

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: \textit{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 $\sqcap$
A type `'a is a **lattice** if

- it is a semilattice
- there is a greatest lower bound operation

\[ \sqcap : \quad 'a \Rightarrow 'a \Rightarrow 'a \]

\[ x \sqcap y \leq x \quad x \sqcap y \leq y \]

\[ [z \leq x; z \leq y] \implies z \leq x \sqcap y \]

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

- \( \text{'av} :: \text{bounded\_lattice} \)
- \( \gamma a_1 \cap \gamma a_2 \subseteq \gamma (a_1 \cap a_2) \)

\[ \implies \gamma (a_1 \cap a_2) = \gamma a_1 \cap \gamma a_2 \]
\[ \implies \cap \text{ is precise!} \]

How about \( \gamma a_1 \cup \gamma a_2 \) and \( \gamma (a_1 \sqcup a_2) \)?

- \( \gamma \bot = \{\} \)
Backward analysis of \( aexp \)

Given

\[
\begin{align*}
  e &::= aexp \\
  a &::= 'av \ (\text{the intended value of } e) \\
  S &::= 'av \ st
\end{align*}
\]

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 \( a \).

What if \( \{ s \in \gamma_S S. \ aval \ e \ s \in \gamma a \} \) is empty?
Work with \( 'av \ st \ option \) instead of \( 'av \ st \)
\textit{inv\_aval}' \ N

\textit{inv\_aval}': \::
\hspace{1cm} aexp \Rightarrow \ 'av \Rightarrow \ 'av \ st \ option \Rightarrow \ 'av \ st \ option

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

An extension of the interface of our framework:

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

Assumption:

\textit{test\_num}'\ i\ a = (i \in \gamma\ a)

Note: \( i \in \gamma\ a \) not necessarily executable
\[ inv_{aval}' (V x) \ a \ S = \]

\[
\text{case } S \text{ of } \text{None} \Rightarrow \text{None} \\
| \text{Some } S \Rightarrow \\
\text{let } a' = \text{fun } S \ x \vartriangle a \\
\text{in if } a' = \bot \text{ then } \text{None} \\
\text{else Some (update } S \ x \ a')
\]

Avoid \( \bot \) component in abstract state, turn abstract state into \text{None} instead.
**inv_aval' Plus**

A further extension of the interface of our framework:

\[ \text{inv_plus'} :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av \]

**Assumption:**

\[ \text{inv_plus'} a a_1 a_2 = (a_1', a_2') \implies \]

\[ \gamma a_1' \supseteq \{ i_1 \in \gamma a_1. \ \exists i_2 \in \gamma a_2. \ i_1 + i_2 \in \gamma a \} \land \]

\[ \gamma a_2' \supseteq \{ i_2 \in \gamma a_2. \ \exists i_1 \in \gamma a_1. \ i_1 + i_2 \in \gamma a \} \]

**Definition:**

\[ \text{inv_aval'} (\text{Plus} \ e_1 \ e_2) a S = \]

\[ (\text{let} (a_1, a_2) = \text{inv_plus'} a (\text{aval''} e_1 S) (\text{aval''} e_2 S) \]

\[ \text{in} \ \text{inv_aval'} e_1 a_1 (\text{inv_aval'} e_2 a_2 S)) \]

(Analogously for all other arithmetic operations)
Backward analysis of \textit{bexp}

Given $b :: bexp$

\textit{res} :: bool (the intended value of $b$)

$S :: 'av st option$

restrict $S$ to some $S' \leq S$ such that

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

$\gamma_s S'$ overapproximates the subset of $\gamma_s S$

that makes $b$ evaluate to $\text{res}$. 
\[\text{inv\_bval'} :: \]
\[\text{bexp} \Rightarrow \text{bool} \Rightarrow 'av \text{ st option} \Rightarrow 'av \text{ st option}\]

\[\text{inv\_bval'} (Bc \ v) \ \text{res} \ S = (\text{if } v = \text{res} \ \text{then } S \ \text{else } \text{None})\]

\[\text{inv\_bval'} (\text{Not} \ b) \ \text{res} \ S = \text{inv\_bval'} b (\neg \ \text{res}) \ S\]

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

\[\text{inv\_bval'} (\text{Less} \ e_1 \ e_2) \ \text{res} \ S =\]
\[\text{let } (a_1, a_2) = \text{inv\_less'} \ \text{res} \ (\text{aval''} \ e_1 \ S) \ (\text{aval''} \ e_2 \ S)\]
\[\text{in } \text{inv\_aval'} e_1 \ a_1 \ (\text{inv\_aval'} e_2 \ a_2 \ S)\]
A further extension of the interface of our framework:

\( inv_{\less}' :: \text{bool} \Rightarrow '\text{av} \Rightarrow '\text{av} \Rightarrow '\text{av} \times '\text{av} \)

Assumption:

\[
\begin{align*}
inv_{\less}' \ res \ a_1 \ a_2 = (a_1', a_2') & \implies \\
\gamma a_1' \supseteq \{ i_1 \in \gamma a_1. \exists i_2 \in \gamma a_2. (i_1 < i_2) = \text{res} \} & \land \\
\gamma a_2' \supseteq \{ i_2 \in \gamma a_2. \exists i_1 \in \gamma a_1. (i_1 < i_2) = \text{res} \}
\end{align*}
\]
Example: intervals, informally

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

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

$\text{inv\_bval}' \ (x + y < z) \ True$

\begin{align*}
\{ & x \mapsto [10, 20], \ y \mapsto [-10, 0], \ z \mapsto [-5, 5] \} : \\
\text{inv\_aval}' \ z \ [1, 5] \ \{ \bullet \} = \{ \bullet, z \mapsto [1, 5] \}
\end{align*}

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

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

$\text{inv\_aval}' \ x \ [10, 14] \ \{ \bullet \} =$

\begin{align*}
\{ & x \mapsto [10, 14], \ y \mapsto [-10, -6], \ z \mapsto [1, 5] \}
\end{align*}
Before: \[ step' = \text{Step asem} (\lambda b \ S. \ S) \]

Now: \[ step' = \text{Step asem} (\lambda b. \text{inv_bval}' \ b \ True) \]
Correctness proof

Almost as before, but with correctness lemmas for \( \text{inv\_aval}' \)

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

and \( \text{inv\_bval}' \):

\[
\{ s \in \gamma_o S. \text{bv} = \text{bval} b s \} \subseteq \gamma_o (\text{inv\_bval}' b \text{bv} S)
\]
Summary

Extended interface to abstract interpreter:

- ′av :: bounded_lattice
  \[ \gamma \bot = \{\} \quad \text{and} \quad \gamma a_1 \cap \gamma a_2 \subseteq \gamma (a_1 \cap a_2) \]

- test_num' :: int ⇒ ′av ⇒ bool
  \[ \text{test_num'} i a = (i \in \gamma a) \]

- inv_plus' :: ′av ⇒ ′av ⇒ ′av ⇒ ′av × ′av
  \[ [\text{inv_plus'} a a_1 a_2 = (a_1', a_2'); \quad i_1 \in \gamma a_1; \quad i_2 \in \gamma a_2; \quad i_1 + i_2 \in \gamma a] \]
  \[ \implies i_1 \in \gamma a_1' \land i_2 \in \gamma a_2' \]

- inv_less' :: bool ⇒ ′av ⇒ ′av ⇒ ′av × ′av
  \[ [\text{inv_less'} (i_1 < i_2) a_1 a_2 = (a_1', a_2'); \quad i_1 \in \gamma a_1; \quad i_2 \in \gamma a_2] \]
  \[ \implies i_1 \in \gamma a_1' \land i_2 \in \gamma a_2' \]
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

- \( x_0 = \bot, \quad x_{i+1} = f(x_i) \)
  - may not terminate while searching for a pfp: \( f(x_i) \leq x_i \)
- Widen in each step: \( x_{i+1} = x_i \triangledown f(x_i) \)
  - until a pfp is found.
- We assume
  - \( \triangledown \) “extrapolates” its arguments: \( x, y \leq x \triangledown y \)
  - \( \triangledown \) “jumps” far enough to prevent nontermination
Example: Widening on (non-empty) intervals

\[ [l_1, h_1] \triangledown [l_2, h_2] = [l, h] \]

where

\[
\begin{align*}
l &= (\text{if } l_1 > l_2 \text{ then } -\infty \text{ else } l_1) \\
h &= (\text{if } h_1 < h_2 \text{ then } \infty \text{ else } h_1)
\end{align*}
\]
Warning

- \( x_{i+1} = f(x_i) \) finds a least pfp if it terminates, \( f \) is monotone, and \( x_0 = \bot \)

- \( x_{i+1} = x_i \nabla f(x_i) \) may return any pfp in the worst case \( \top \)

We win termination, we lose precision
iteration of $f$
iteration with widening
narrowing iteration of $f$
A widening operator $\triangledown :: 'a \Rightarrow 'a \Rightarrow 'a$ on a preorder must satisfy $x \leq x \triangledown y$ and $y \leq x \triangledown y$.

Widening operators can be extended from $'a$ to $'a$ st, $'a$ option and $'a$ acom.
Abstract interpretation with widening

New assumption: \( \texttt{\text{\textquotesingle}av \text{ has widening operator}} \)

\[
\text{iter\_widen :: (}'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ option}
\]

\[
\text{iter\_widen } f = \text{while\_option } (\lambda x. \neg f x \leq x) (\lambda x. x \triangledown f x)
\]

Correctness (returns pfp): by definition

Abstract interpretation of \( c \):

\[
\text{iter\_widen (step' } \bot) (\text{bot } c)
\]
\( x ::= N\ 0 \{ A_0 \} ;; \)
\( \{ A_1 \} \)
\( WHILE \ Less \ ( V \ x ) \ ( N \ 100 ) \)
\( DO \ \{ A_2 \} \)
\( x ::= Plus \ ( V \ x ) \ ( N \ 1 ) \ \{ A_3 \} \)
\( \{ A_4 \} \)
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 $\triangle :: \mathbf{'a} \Rightarrow \mathbf{'a} \Rightarrow \mathbf{'a}$ must satisfy $y \leq x \Rightarrow y \leq x \triangle y \leq x$.

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

```plaintext
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

\[ l = (if \ l_1 = -\infty \ then \ l_2 \ else \ l_1) \]
\[ h = (if \ h_1 = \infty \ then \ h_2 \ else \ h_1) \]
Abstract interpretation with widening & narrowing

New assumption: 'av also has a narrowing operator

\[ \text{pfp}_\text{wn} f x = (\text{case } \text{iter}_\text{widen} f x \text{ of } \text{None } \Rightarrow \text{None } \mid \text{Some } p \Rightarrow \text{iter}_\text{narrow} f p) \]

\[ \text{AI}_\text{wn} c = \text{pfp}_\text{wn} (\text{step}' \top) (\text{bot } c) \]

**Theorem** \( \text{AI}_\text{wn} c = \text{Some } C \implies CS c \leq \gamma_c C \)

**Proof** as before
Termination

of

\[ \text{while\_option} \ (\lambda x. \ P \ x) \ (\lambda x. \ g \ x) \]

via measure function \( m \)

such that \( m \) goes down with every iteration:

\[ P \ x \implies m \ x > m(g \ x) \]

May need some invariant \( Inv \) as additional premise:

\[ Inv \ x \implies P \ x \implies m \ x > m(g \ x) \]
Termination of \textit{iter\_widen}

\begin{align*}
\text{iter\_widen } f &= \\
\text{while\_option } (\lambda x. \neg f x \leq x) (\lambda x. x \triangledown f x)
\end{align*}

As before (almost): Assume \( m :: \text{'av} \Rightarrow \text{nat} \) and \( h :: \text{nat} \)
such that \( m x \leq h \) and \( x \leq y \implies m y \leq m x \)
and additionally \( \neg y \leq x \implies m (x \triangledown y) < m x \)

Define the same functions \( m_s/m_o/m_c \) as before.

Termination of \textit{iter\_widen} on \textit{'}a st option acom:\n
\textbf{Lemma} \( \neg C_2 \leq C_1 \implies m_c (C_1 \triangledown C_2) < m_c C_1 \)

if \( \text{top\_on}_c C_1 (\neg \text{vars } C_1) \), \( \text{top\_on}_c C_2 (\neg \text{vars } C_2) \)
and \( \text{strip } C_1 = \text{strip } C_2 \)
Termination of \textit{iter\_narrow}

\texttt{iter\_narrow} f =
\texttt{while\_option} \ (\lambda x. \ x \triangle f \ x < x) \ (\lambda x. \ x \triangle f \ x)

Assume \( n :: 'av \Rightarrow \text{nat} \) such that
\[ [y \leq x; x \triangle y < x] \implies n \ (x \triangle y) < n \ x \]

Define \( n_s/n_o/n_c \) like \( m_s/m_o/m_c \)

Termination of \textit{iter\_narrow} on \('a \ st \ option \ acom\):
\textbf{Lemma} \[ [ \ C_2 \leq C_1; \ C_1 \triangle C_2 < C_1 \ ] \implies \]
\( n_c \ (C_1 \triangle C_2) < n_c \ C_1 \) \ if \ strip \( C_1 = \text{strip} \ C_2 \),
\( \text{top\_on}_c \ C_1 \ (\neg \ \text{vars} \ C_1) \) \ and \( \text{top\_on}_c \ C_2 \ (\neg \ \text{vars} \ C_2) \)
Measuring non-empty intervals

\[ m [l,h] = (\text{if } l = -\infty \text{ then } 0 \text{ else } 1) + (\text{if } h = \infty \text{ then } 0 \text{ else } 1) \]

\[ h = 2 \]

\[ n \text{ ivl} = 2 - m \text{ ivl} \]