Concrete Semantics with Isabelle/HOL

Jasmin Blanchette\textsuperscript{1,2}  Mathias Fleury\textsuperscript{2}  Daniel Wand\textsuperscript{2}

\textsuperscript{1} Inria Nancy & LORIA  \textsuperscript{2} Max-Planck-Institut für Informatik

based on slides by Tobias Nipkow  
2015-08-4
Chapter 1

Introduction
1 Background

2 This Course
1 Background

2 This Course
Why Semantics?

Without semantics, we do not really know what our programs mean. We merely have a good intuition and a warm feeling. Like the state of mathematics in the 19th century — before set theory and logic entered the scene.
Intuition is important!

- You need a good intuition to get your work done efficiently.
- To understand the average accounting program, intuition suffices.
- To write a bug-free accounting program may require more than intuition!
- We assume you have the necessary intuition.
- This course is about “beyond intuition”.
Intuition is not sufficient!

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

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

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

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

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

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

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

GI Dissertationspreis 2003:
Gerwin Klein: *Verified Java Bytecode Verification*
Standard ML (SML)


Main achievements: LCF (theorem proving) SML (functional programming) CCS, pi (concurrency)
The sad fact of life

SML semantics hardly used:

- too difficult to read to answer simple questions quickly
- too much detail to allow reliable informal proof
- not processable beyond \LaTeX, not even executable
More sad facts of life

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

Machine-checked language semantics and proofs

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

The tool:

Proof Assistant (PA)

or

Interactive Theorem Prover (ITP)
Proof Assistants

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

Government health warnings:

- Time consuming
- Potentially addictive
- Undermines your naive trust in informal proofs
This lecture course:

\[ \text{Formal} = \text{machine-checked} \]
\[ \text{Verification} = \text{formal correctness proof} \]

Traditionally:

\[ \text{Formal} = \text{mathematical} \]
Two landmark verifications

C compiler
Competitive with gcc -O1

Xavier Leroy
INRIA Paris
using Coq

Operating system
microkernel (L4)

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle
A happy fact of life

Programming language researchers are increasingly using PAs
Why verification pays off

Short term: *The software works!*

Long term:

- Tracking effects of changes by rerunning proofs
- Incremental changes of the software typically require only incremental changes of the proofs

Long term much more important than short term:

*Software Never Dies*
1 Background

2 This Course
What this course is *not* about

- Hot or trendy PLs
- Comparison of PLs or PL paradigms
- Compilers (although they will be one application)
What this course *is* about

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

Both informally and formally (PA!)
Our PA: Isabelle/HOL

- Started 1986 in Cambridge (Paulson)
- Later development mainly in Munich (Nipkow & Co.)
- The logic HOL is ordinary mathematics

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

All exercises require the use of Isabelle/HOL
Why do we care about the PA part

- It is the future
- It is the only way to deal with complex languages reliably
- It teaches students how to write correct proofs
- Too many proofs look more like LSD trips than coherent mathematical arguments
Overview of course

- Introduction to Isabelle/HOL
- IMP (assignment and while loops) and its semantics
- A compiler for IMP
- Hoare logic for IMP
- Type systems for IMP
- Program analysis for IMP
The semantics part of the course is mostly traditional.

The use of a PA is leading edge.

A growing number of universities offer related course.
What you learn in this course goes far beyond PLs

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

It is a new approach to informatics
Part I

Isabelle
Chapter 2

Programming and Proving
3 Overview of Isabelle/HOL
4 Type and function definitions
5 Induction Heuristics
6 Simplification
Implication associates to the right:

\[ A \implies B \implies C \quad \text{means} \quad A \implies (B \implies C) \]

Similarly for other arrows: \( \Rightarrow \), \( \implies \)

\[ \frac{A_1 \ldots A_n}{B} \quad \text{means} \quad A_1 \implies \ldots \implies A_n \implies B \]
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

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

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:
- For the moment: only term = term,
e.g. 1 + 2 = 4
- Later: ∧, ∨, →, ∀, ...
Overview of Isabelle/HOL

Types and terms

Interface

By example: Types \textit{bool}, \textit{nat}, and \textit{list}

Summary
Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \mid \text{type variables} \]

\[ \tau \Rightarrow \tau \mid \tau \times \tau \mid \tau \text{ list} \mid \tau \text{ set} \mid \ldots \mid \text{base types} \]

Convention:

\[ \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3) \]
Terms

Terms can be formed as follows:

- **Function application**: $f \ t$
  is the call of function $f$ with argument $t$.
  If $f$ has more arguments: $f \ t_1 \ t_2 \ldots$
  Examples: $\sin \ \pi$, $\text{plus} \ x \ y$

- **Function abstraction**: $\lambda x. \ t$
  is the function with parameter $x$ and result $t$, i.e. "$x \mapsto \ t$".
  Example: $\lambda x. \ \text{plus} \ x \ x$
Terms

Basic syntax:

\[
t ::= (t) \\
| \ a \quad \text{constant or variable (identifier)} \\
| \ tt \quad \text{function application} \\
| \ \lambda x. \ t \quad \text{function abstraction} \\
| \ \ldots \quad \text{lots of syntactic sugar}
\]

Examples:

\[
f (g \ x) \ y \\
h (\lambda x. \ f (g \ x))
\]

Convention:

\[
f t_1 \ t_2 \ t_3 \equiv ((f \ t_1) \ t_2) \ t_3
\]

This language of terms is known as the \(\lambda\)-calculus.
The computation rule of the $\lambda$-calculus is the replacement of formal by actual parameters:

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

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

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

- The step from $(\lambda x. \ t) \ u$ to $t[u/x]$ is called $\beta$-reduction.
- Isabelle performs $\beta$-reduction automatically.
Terms must be well-typed
(the argument of every function call must be of the right type)

Notation:
\( t :: \tau \) means "\( t \) is a well-typed term of type \( \tau \)".

\[
\frac{t :: \tau_1 \Rightarrow \tau_2}{t \ u :: \tau_2}
\]

\[
\frac{u :: \tau_1}{t \ u :: \tau_2}
\]
Isabelle automatically computes the type of each variable in a term. This is called \textit{type inference}.

In the presence of \textit{overloaded} functions (functions with multiple types) this is not always possible.

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

Thou shalt Curry thy functions

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

Advantage:

Currying allows *partial application*

\[ f \ a_1 \ \text{where} \ a_1 :: \tau_1 \]
Predefined syntactic sugar

- **Infix:** +, −, *, #, @, ...
- **Mixfix:** if _ then _ else _, case _ of_, ...

Prefix binds more strongly than infix:

$$f x + y \equiv (f x) + y \neq f (x + y)$$

Enclose *if* and *case* in parentheses:

$$(! (if _ then _ else _))$$
Theory = Isabelle Module

Syntax:

```
theory MyTh
imports T_1 \ldots T_n
begin
(definitions, theorems, proofs, \ldots)^*
end
```

*MyTh*: name of theory. Must live in file *MyTh.thy*

*Ti*: names of *imported* theories. Import transitive.

Usually:  

```
imports Main
```
Concrete syntax

In .thy files:
Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides
Overview of Isabelle/HOL

Types and terms

Interface

By example: Types $\text{bool}$, $\text{nat}$, and $\text{list}$

Summary
isabelle jedit

- Based on *jedit* editor
- Processes Isabelle text automatically when editing `.thy` files (like modern Java IDEs)
Overview_Demo.thy
Overview of Isabelle/HOL

Types and terms

By example: Types bool, nat, and list

Summary
Type \texttt{bool}

datatype \texttt{bool} = True \mid False

Predefined functions:
\land, \lor, \rightarrow, \ldots \implies bool \Rightarrow bool \Rightarrow bool

A formula is a term of type \texttt{bool}

if-and-only-if: =
Type \textit{nat}

\textbf{datatype} \hspace{1em} \textit{nat} = 0 \mid \textit{Suc} \textit{nat}

Values of type \textit{nat}: 0, \textit{Suc} 0, \textit{Suc}(\textit{Suc} 0), \ldots

Predefined functions: +, *, ... \:: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}

\textbf{!} Numbers and arithmetic operations are overloaded:

\hspace{1em} 0,1,2,... \:: \textit{'}a, \hspace{1em} + \:: \textit{'}a \Rightarrow \textit{'}a \Rightarrow \textit{'}a

You need type annotations: 1 :: \textit{nat}, x + (y::\textit{nat})

unless the context is unambiguous: \textit{Suc} z
Nat_Demo.thy
**Lemma** \( add \ m \ 0 = m \)

**Proof** by induction on \( m \).

- **Case** 0 (the base case):
  \( add \ 0 \ 0 = 0 \) holds by definition of \( add \).

- **Case** \( Suc \ m \) (the induction step):
  We assume \( add \ m \ 0 = m \), the induction hypothesis (IH).
  We need to show \( add \ (Suc \ m) \ 0 = Suc \ m \).
  The proof is as follows:

\[
add \ (Suc \ m) \ 0 = Suc \ (add \ m \ 0) \quad \text{by def. of} \ add
\]

\[
= Suc \ m \quad \text{by IH}
\]
Type 'a list

Lists of elements of type 'a

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

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

Syntactic sugar:
- [] = Nil: empty list
- x # xs = Cons x xs:
  list with first element x ("head") and rest xs ("tail")
- [x₁, ..., xₙ] = x₁ # ... # xₙ # []
Structural Induction for lists

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

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

\[
\frac{P([]) \land \forall x \; xs. \; P(xs) \implies P(x\#xs)}{P(xs)}
\]
List_Demo.thy
An informal proof

**Lemma** \( \text{app} \ (\text{app} \ xs \ ys) \ zs = \text{app} \ xs \ (\text{app} \ ys \ zs) \)

**Proof** by induction on \( xs \).

- Case \( \text{Nil} \): \( \text{app} \ (\text{app} \ \text{Nil} \ ys) \ zs = \text{app} \ ys \ zs = \text{app} \ \text{Nil} \ (\text{app} \ ys \ zs) \) holds by definition of \( \text{app} \).

- Case \( \text{Cons} \ x \ xs \): We assume \( \text{app} \ (\text{app} \ xs \ ys) \ zs = \text{app} \ xs \ (\text{app} \ ys \ zs) \) (IH), and we need to show \( \text{app} \ (\text{app} \ (\text{Cons} \ x \ xs) \ ys) \ zs = \text{app} \ (\text{Cons} \ x \ xs) \ (\text{app} \ ys \ zs) \).

The proof is as follows:

\[
\begin{align*}
\text{app} \ (\text{app} \ (\text{Cons} \ x \ xs) \ ys) \ zs \\
= \text{Cons} \ x \ (\text{app} \ (\text{app} \ xs \ ys) \ zs) \quad \text{by definition of} \ \text{app} \\
= \text{Cons} \ x \ (\text{app} \ xs \ (\text{app} \ ys \ zs)) \quad \text{by} \ \text{IH} \\
= \text{app} \ (\text{Cons} \ x \ xs) \ (\text{app} \ ys \ zs) \quad \text{by definition of} \ \text{app}
\end{align*}
\]
Large library: HOL/List.thy

Included in Main.

Don’t reinvent, reuse!

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

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

fun $map :: (\texttt{'}a \Rightarrow \texttt{'}b) \Rightarrow \texttt{'}a \ \texttt{list} \Rightarrow \texttt{'}b \ \texttt{list}$ where

$map \ f \ [] = []$

$map \ f \ (x\#xs) = f \ x \# \ map \ f \ xs$

Note: $map$ takes function as argument.
Overview of Isabelle/HOL

Types and terms

Interface

By example: Types bool, nat, and list

Summary
• **datatype** defines (possibly) recursive data types.

• **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.
Proof methods

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

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

  “=” is used only from left to right!
Proofs

General schema:

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

If the lemma is suitable as a simplification rule:

```
lemma name[simp]: "..."
```
Top down proofs

Command

\textbf{sorry}

“completes” any proof.

Allows top down development:

\textit{Assume lemma first, prove it later.}
The proof state

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

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

\[
\left[ A_1; \ldots ; A_n \right] \implies B
\]

abbreviates

\[ A_1 \implies \ldots \implies A_n \implies B \]

; \approx \text{“and”}
Overview of Isabelle/HOL

Type and function definitions

Induction Heuristics

Simplification
4 Type and function definitions

Type definitions

Function definitions
Type synonyms

**type_synonym**  name $= \tau$

Introduces a *synonym* name for type $\tau$

Examples:

**type_synonym**  string $= \text{char list}$

**type_synonym**  ($'a,'b)$foo $= 'a \text{ list} \times 'b \text{ list}$

Type synonyms are expanded after parsing and are not present in internal representation and output
**datatype — the general case**

\[
\text{datatype } (\alpha_1, \ldots, \alpha_n)\tau = \begin{array}{c}
C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\vdots \\
C_k \tau_{k,1} \ldots \tau_{k,n_k}
\end{array}
\]

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

Distinctness and injectivity are applied automatically. Induction must be applied explicitly.
Case expressions

Datatype values can be taken apart with `case`:

\[
(case \, xs \, of \, \begin{array}{l}
\quad [] \Rightarrow \ldots \\
\quad \_ \Rightarrow \ldots \; y \ldots \; ys \ldots 
\end{array})
\]

Wildcards: _

\[
(case \, m \, of \, 0 \Rightarrow Suc \, 0 \quad | \quad Suc \, _ \Rightarrow 0)
\]

Nested patterns:

\[
(case \, xs \, of \, \begin{array}{l}
\quad [0] \Rightarrow 0 \\
\quad [Suc \, n] \Rightarrow n \\
\quad _ \Rightarrow 2
\end{array})
\]

Complicated patterns mean complicated proofs!

Need ( ) in context
Tree_Demo.thy
The *option* type

```datatype 'a option = None | Some 'a
```

If `'a` has values `a_1`, `a_2`, ... 
then `'a option` has values `None`, `Some a_1`, `Some a_2`, ...

Typical application:

```fun lookup :: ('a × 'b) list ⇒ 'a ⇒ 'b option where
lookup [] x = None |
lookup ((a,b) # ps) x =
  (if a = x then Some b else lookup ps x)
```
4 Type and function definitions

Type definitions

Function definitions
Non-recursive definitions

Example:

**definition** \( sq :: \text{nat} \Rightarrow \text{nat} \text{ where } sq \ n = n \times n \)

No pattern matching, just \( f \ x_1 \ldots \ x_n = \ldots \)
The danger of nontermination

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

All functions in HOL must be total!
Key features of \textbf{fun}

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

fun sep :: 'a ⇒ 'a list ⇒ 'a list where
sep a (x#y#zs) = x # a # sep a (y#zs) | 
sep a xs = xs
Example: Ackermann

fun ack :: nat ⇒ nat ⇒ nat where
ack 0 n = Suc n |
ack (Suc m) 0 = ack m (Suc 0) |
ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

Terminates because the arguments decrease lexicographically with each recursive call:

- \((\text{Suc } m, 0) > (m, \text{Suc } 0)\)
- \((\text{Suc } m, \text{Suc } n) > (\text{Suc } m, n)\)
- \((\text{Suc } m, \text{Suc } n) > (m, _)\)
A restrictive version of `fun`
- Means `primitive recursive`
- Most functions are primitive recursive
- Frequently found in Isabelle theories

The essence of primitive recursion:

\[
\begin{align*}
f(0) & = \ldots \text{ no recursion} \\
f(Suc \ n) & = \ldots f(n) \ldots \\
g([]) & = \ldots \text{ no recursion} \\
g(x # xs) & = \ldots g(xs) \ldots 
\end{align*}
\]
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification
Basic induction heuristics

Theorems about recursive functions are proved by induction.

Induction on argument number $i$ of $f$ if $f$ is defined by recursion on argument number $i$. 
A tail recursive reverse

Our initial reverse:

fun rev :: 'a list ⇒ 'a list where
  rev [] = [] |
  rev (x#xs) = rev xs @ [x]

A tail recursive version:

fun itrev :: 'a list ⇒ 'a list ⇒ 'a list where
  itrev [] ys = ys |
  itrev (x#xs) ys =

lemma itrev xs [] = rev xs
Induction_Demo.thy

Generalisation
Generalisation

- Replace constants by variables
- Generalize free variables
  - by *arbitrary* in induction proof
  - (or by universal quantifier in formula)
So far, all proofs were by structural induction because all functions were primitive recursive.

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

Now: induction for complex recursion patterns.
**Computation Induction: Example**

```plaintext
fun div2 :: nat ⇒ nat where

div2 0 = 0  |

div2 (Suc 0) = 0  |

div2 (Suc (Suc n)) = Suc (div2 n)
```

∼ induction rule div2.induct:

$$
\begin{align*}
P(0) & \quad P(Suc \ 0) \quad \land \ n. \ P(n) & \quad \Longrightarrow \ P(Suc(Suc \ n)) \\
\hline \\
& \quad P(m)
\end{align*}
$$
Computation Induction

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

for each defining equation

\[
f(e) = \ldots f(r_1) \ldots f(r_k) \ldots
\]

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

Induction follows course of (terminating!) computation

Motto: properties of \( f \) are best proved by rule \textit{f.induct}
How to apply \texttt{f.induct}

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

$(\textit{induction } a_1 \ldots a_n \text{ rule: } f\texttt{.induct})$

Heuristic:

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

Computation Induction
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification
Simplification means . . .

Using equations \( l = r \) from left to right

As long as possible

Terminology: equation \( \sim \) simplification rule

Simplification = (Term) Rewriting
An example

Equations:

\[ 0 + n = n \] \hspace{1cm} \text{(1)}
\[ (\text{Suc } m) + n = \text{Suc } (m + n) \] \hspace{1cm} \text{(2)}
\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \] \hspace{1cm} \text{(3)}
\[ (0 \leq m) = \text{True} \] \hspace{1cm} \text{(4)}

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \] \hspace{1cm} \text{(1)} \quad \equiv \quad \text{(1)}
\[ \text{Suc } 0 \leq \text{Suc } 0 + x \] \hspace{1cm} \text{(2)} \quad \equiv \quad \text{(2)}
\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \] \hspace{1cm} \text{(3)} \quad \equiv \quad \text{(3)}
\[ 0 \leq 0 + x \] \hspace{1cm} \text{(4)} \quad \equiv \quad \text{(4)}
\[ \text{True} \]
Conditional rewriting

Simplification rules can be conditional:

\[
[ P_1; \ldots; P_k ] \implies l = r
\]

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

Example:

\[
p(0) = True
\]

\[
p(x) \implies f(x) = g(x)
\]

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

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

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

\[
\begin{bmatrix} P_1; \ldots; P_k \end{bmatrix} \implies l = r
\]

is suitable as a simp-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[
n < m \implies (n < \text{Suc} \ m) = \text{True} \quad \text{YES}
\]

\[
\text{Suc} \ n < m \implies (n < m) = \text{True} \quad \text{NO}
\]
Proof method *simp*

Goal: 1. \[ [ P_1; \ldots; P_m ] \implies C \]

**apply** (*simp add: eq_1 \ldots eq_n*)

Simplify \( P_1 \ldots P_m \) and \( C \) using

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

Variations:

- \((simp \ldots del: \ldots)\) removes *simp*-lemmas
- *add* and *del* are optional
\textit{auto} versus \textit{simp}

- \textit{auto} acts on all subgoals
- \textit{simp} acts only on subgoal 1
- \textit{auto} applies \textit{simp} and more
- \textit{auto} can also be modified:
  \begin{verbatim}
  (auto simp add: ... simp del: ...)
  \end{verbatim}
Rewriting with definitions

Definitions (definition) must be used explicitly:

\[(\text{simp add: } f\_\text{def} \ldots)\]

\(f\) is the function whose definition is to be unfolded.
Case splitting with \textit{simp}

Automatic:

\[ P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t)) \]

By hand:

\[ P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \ e = \text{Suc } n \rightarrow P(b)) \]

Proof method: (\textit{simp split: nat.split})
Or \textit{auto}. Similar for any datatype \( t \): \( t.split \)
Simp_Demo.thy
Chapter 3

Case Study: IMP Expressions
Case Study: IMP Expressions
7 Case Study: IMP Expressions
This section introduces

*arithmetic and boolean expressions*

of our imperative language IMP.

IMP *commands* are introduced later.
Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
Concrete and abstract syntax

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

Abstract syntax: trees, eg

```
+  
/ 
a *  
   / 
   5 b
```

Parser: function from strings to trees

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

Abstract syntax trees/terms are datatype values!
Concrete syntax is defined by a context-free grammar, eg

\[ a ::= n \mid x \mid (a) \mid a + a \mid a \ast a \mid \ldots \]

where \( n \) can be any natural number and \( x \) any variable.

We focus on abstract syntax which we introduce via datatypes.
Datatype $aexp$

Variable names are strings, values are integers:

**type_synonym** $vname = string$

**datatype** $aexp = N int | V vname | Plus aexp aexp$

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$N 5$</td>
</tr>
<tr>
<td>$x$</td>
<td>$V &quot;x&quot;$</td>
</tr>
<tr>
<td>$x+y$</td>
<td>$Plus (V &quot;x&quot;) (V &quot;y&quot;)$</td>
</tr>
<tr>
<td>$2+(z+3)$</td>
<td>$Plus (N 2) (Plus (V &quot;z&quot;) (N 3))$</td>
</tr>
</tbody>
</table>
Warning

This is syntax, not (yet) semantics!

\( N \ 0 \ \neq \ Plus \ (N \ 0) \ (N \ 0) \)
The (program) state

What is the value of \( x+1 \)?

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

\[
\begin{align*}
\text{type_synonym } & \text{ val } = \text{ int } \\
\text{type_synonym } & \text{ state } = \text{ vname } \Rightarrow \text{ val }
\end{align*}
\]
Function update notation

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

$$f(a := b)$$

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

$$f(a := b) = (\lambda x. \text{if } x = a \text{ then } b \text{ else } f x)$$
How to write down a state

Some states:

- \( \lambda x. 0 \)
- \( (\lambda x. 0)("a" := 3) \)
- \( ((\lambda x. 0)("a" := 5))("x" := 3) \)

Nicer notation:

\[ <"a" := 5, "x" := 3, "y" := 7> \]

Maps everything to 0, but "a" to 5, "x" to 3, etc.
AExp.thy
Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
BExp.thy
Case Study: IMP Expressions

- Arithmetic Expressions
- Boolean Expressions

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

We need more logical machinery to define program execution and reason about it.
Chapter 4

Logic and Proof
Beyond Equality
8 Logical Formulas

9 Proof Automation

10 Single Step Proofs

11 Inductive Definitions
Logical Formulas

Proof Automation

Single Step Proofs

Inductive Definitions
Syntax (in decreasing precedence):

\[
form ::= \ (form) \quad | \quad term = term \quad | \quad \neg form
\]
\[
\quad | \quad form \ \land \ form \quad | \quad form \ \lor \ form \quad | \quad form \rightarrow form
\]
\[
\quad | \quad \forall x. \ form \quad | \quad \exists x. \ form
\]

Examples:

\[
\neg A \land B \lor C \equiv ((\neg A) \land B) \lor C
\]
\[
s = t \land C \equiv (s = t) \land C
\]
\[
A \land B = B \land A \equiv A \land (B = B) \land A
\]
\[
\forall x. \ P \ x \land Q \ x \equiv \forall x. (P \ x \land Q \ x)
\]

Input syntax: \(\longleftrightarrow\) (same precedence as \(\rightarrow\))
Variable binding convention:

\[ \forall x \ y. \ P \ x \ y \equiv \ \forall x. \ \forall y. \ P \ x \ y \]

Similarly for \( \exists \) and \( \lambda \).
Warning

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

\[ P \land \forall x. Q x \quad \leadsto \quad P \land (\forall x. Q x) \]
Mathematical symbols

... and their ASCII representations:

\(\forall\quad <\text{forall}>\quad \text{ALL}\)

\(\exists\quad <\text{exists}>\quad \text{EX}\)

\(\lambda\quad <\text{lambda}>\quad \%\)

\(\longrightarrow\quad -->\)

\(\longleftrightarrow\quad <--\)

\(\land\quad \land\quad \&\)

\(\lor\quad \lor\quad |\)

\(\neg\quad <\text{not}>\quad ~\)

\(\neq\quad <\text{noteq}>\quad \sim=\)
Sets over type \(\texttt{'a}\)

\(\texttt{'a set}\)

- \{\}, \{e_1, \ldots, e_n\}
- \(e \in A, A \subseteq B\)
- \(A \cup B, A \cap B, A - B, -A\)
- \ldots

\(\in \ \lt\text{in}\gt\) : \(\subseteq \ \lt\text{subseteq}\gt \\leq\)

\(\cup \ \lt\text{union}\gt \text{ Un}\)

\(\cap \ \lt\text{inter}\gt \text{ Int}\)
Set comprehension

- \( \{ x. \ P \} \) where \( x \) is a variable
- But not \( \{ t. \ P \} \) where \( t \) is a proper term
- Instead: \( \{ t \mid x \ y \ z. \ P \} \) is short for \( \{ v. \ \exists x \ y \ z. \ v = t \land P \} \) where \( x, \ y, \ z \) are the free variables in \( t \)
8 Logical Formulas
9 Proof Automation
10 Single Step Proofs
11 Inductive Definitions
simp and auto

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

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

Exception: auto acts on all subgoals
fastforce

- rewriting, logic, sets, relations and a bit of arithmetic.
- incomplete but better than auto.
- Succeeds or fails
- Extensible with new simp-rules
• A **complete** proof search procedure for FOL . . .
• . . . but (almost) **without** “=”
• Covers logic, sets and relations
• Succeeds or fails
• Extensible with new deduction rules
Automating arithmetic

\textit{arith}:

- proves linear formulas (no \textquotedblleft{*}\textquotedblright)
- complete for quantifier-free \textit{real} arithmetic
- complete for first-order theory of \textit{nat} and \textit{int}
  (Presburger arithmetic)
Sledgehammer
Architecture:

Isabelle

Goal & filtered library

\[\downarrow \quad \uparrow\]

external

Proof

\textbf{ATPs}^{1}

Characteristics:

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

Do you feel lucky?

\[^{1}\text{Automatic Theorem Provers} \]
by \((proof\text{-}method)\)

\[\approx\]

apply \((proof\text{-}method)\)
done
Auto_Proof_Demo.thy
8 Logical Formulas
9 Proof Automation
10 Single Step Proofs
11 Inductive Definitions
Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed by taking the goal apart.
What are these ?-variables ?

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

Example: theorem conjI: \( [[?P; ?Q]] \rightarrow ?P \land ?Q \)

These ?-variables can later be instantiated:

- By hand:
  
  \[
  \text{conjI[of "a=b" "False"] \leadsto [a = b; False] \rightarrow a = b \land False}
  \]

- By unification:
  
  unifying \( ?P \land ?Q \) with \( a=b \land False \)
  
  sets \( ?P \) to \( a=b \) and \( ?Q \) to \( False \).
Rule application

Example: rule: \[ ?P; ?Q \] \implies ?P \land ?Q

subgoal: 1. \ldots \implies A \land B

Result: 1. \ldots \implies A
2. \ldots \implies B

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

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

apply(rule xyz)

“Backchaining”
Typical backwards rules

\[
\begin{align*}
\frac{?P \quad ?Q}{?P \land ?Q} & \quad \text{conjI} \\
\frac{?P \quad \forall x. ?P x}{\forall x. ?P x} & \quad \text{allI} \\
\frac{?P \quad ?Q \quad ?Q \quad ?P}{?P \equiv ?Q} & \quad \text{iffI}
\end{align*}
\]

They are known as introduction rules because they introduce a particular connective.
Teaching \textit{blast} new intro rules

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

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

allows \textit{blast} to backchain on $r$ during proof search.

Example:

\begin{align*}
\text{theorem } & \textit{trans: } [ \ ?x \leq \ ?y; \ ?y \leq \ ?z \ ] \implies \ ?x \leq \ ?z \\
\text{goal } & 1. [ \ a \leq b; \ b \leq c; \ c \leq d \ ] \implies a \leq d \\
\text{proof } & \textbf{apply}(\text{blast intro: trans})
\end{align*}

Can greatly increase the search space!
Forward proof: OF

If $r$ is a theorem $A \implies B$ and $s$ is a theorem that unifies with $A$ then

$$r[\text{OF } s]$$

is the theorem obtained by proving $A$ with $s$.

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

$$\text{conjI}[\text{OF refl[of "a"]}]$$

$$\leadsto$$

$$\ ?Q \implies a = a \land \ ?Q$$
The general case:

If \( r \) is a theorem \([ A_1; \ldots; A_n ] \imp A\) and \( r_1, \ldots, r_m \ (m \leq n) \) are theorems then

\[ r[\text{OF} \; r_1 \ldots \; r_m] \]

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

Example: theorem refl: \(?t = ?t\)

\[
\text{conjI}[\text{OF} \; \text{refl[of "a"]} \; \text{refl[of "b"]}] \\
\sim \rightarrow \\
\quad a = a \land b = b
\]
From now on: ? mostly suppressed on slides
Single_Step_Demo.thy
is part of the Isabelle framework. It structures theorems and proof states: \([ A_1; \ldots; A_n ] \Rightarrow A\)

is part of HOL and can occur inside the logical formulas \(A_i\) and \(A\).

Phrase theorems like this \([ A_1; \ldots; A_n ] \Rightarrow A\)
not like this \(A_1 \land \ldots \land A_n \Rightarrow A\)
8 Logical Formulas
9 Proof Automation
10 Single Step Proofs
11 Inductive Definitions
Example: even numbers

Informally:

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

In Isabelle/HOL:

```isabelle
inductive ev :: nat ⇒ bool where
  ev 0
| ev n ⇒ ev (n + 2)
```
An easy proof: \( ev 4 \)

\[
ev 0 \implies ev 2 \implies ev 4
\]
Consider

**fun even :: nat ⇒ bool where**

\[
even \ 0 = \ True \mid
\]
\[
even \ (\text{Suc} \ 0) = \ False \mid
\]
\[
even \ (\text{Suc} \ (\text{Suc} \ n)) = \ even \ n
\]

A trickier proof: \( ev \ m \implies even \ m \)

By induction on the *structure* of the derivation of \( ev \ m \)

Two cases: \( ev \ m \) is proved by

- **rule** \( ev \ 0 \)
  \[
  \implies m = 0 \implies even \ m = True
  \]

- **rule** \( ev \ n \implies ev \ (n+2) \)
  \[
  \implies m = n+2 \text{ and } even \ n \ (\text{IH})
  \]
  \[
  \implies even \ m = even \ (n+2) = even \ n = True
  \]
Rule induction for \( ev \)

To prove

\[ ev \ n \implies P \ n \]

by \textit{rule induction} on \( ev \ n \) we must prove

- \( P \ 0 \)
- \( P \ n \implies P(n+2) \)

Rule ev.induct:

\[
\begin{array}{c}
  ev \ n \quad P \ 0 \\
  \wedge n. \ [ ev \ n; \ P \ n ] \implies P(n+2) \\
  P \ n \\
\end{array}
\]
Format of inductive definitions

\textbf{inductive} \( I :: \tau \Rightarrow \text{bool} \ \text{where} \)
\[
[ I \ a_1; \ldots ; I \ a_n ] \implies I \ a
\]

\begin{itemize}
  \item \( I \) may have multiple arguments.
  \item Each rule may also contain \textit{side conditions} not involving \( I \).
\end{itemize}
Rule induction in general

To prove

$I x \implies P x$

by rule induction on $I x$

we must prove for every rule

$$[[ I \ a_1; \ldots; \ I \ a_n ]] \implies I \ a$$

that $P$ is preserved:

$$[[ I \ a_1; \ P \ a_1; \ldots; \ I \ a_n; \ P \ a_n ]] \implies P \ a$$
Rule induction is absolutely central to (operational) semantics and the rest of this lecture course.
Inductive_Demo.thy
Inductively defined sets

\textbf{inductive\_set} \: I :: \tau \: \textit{set} \: \textbf{where} \\
\left[ \: a_1 \in I; \: \ldots \: ; \: a_n \in I \: \right] \implies a \in I \\
\textbf{:} \\

Difference to \textbf{inductive}: \\
\begin{itemize}
\item arguments of \( I \) are tupled, not curried \\
\item \( I \) can later be used with set theoretic operators, eg \( I \cup \ldots \) 
\end{itemize}
Chapter 5

Isar: A Language for Structured Proofs
12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction
Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!
Apply scripts versus Isar proofs

Apply script = assembly language program
Isar proof = structured program with comments

But: apply still useful for proof exploration
A typical Isar proof

proof
  assume $\text{formula}_0$
  have $\text{formula}_1$ by simp
  
  have $\text{formula}_n$ by blast
  show $\text{formula}_{n+1}$ by \ldots

qed

proves $\text{formula}_0 \implies \text{formula}_{n+1}$
Isar core syntax

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

method = (simp ...) | (blast ...) | (induction ...) | ...

step = fix variables (∧)
   | assume prop (⇒)
   | [from fact+] (have | show) prop proof

prop = [name:] “formula”

fact = name | ...
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Example: Cantor’s theorem

**lemma** \( \neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set}) \)

**proof** default proof: assume \( \text{surj} \), show \( \text{False} \)

assume \( a: \text{surj} f \)

from \( a \) have \( b: \forall A. \exists a. A = f a \)

by (simp add: surj_def)

from \( b \) have \( c: \exists a. \{ x. x \notin f x \} = f a \)

by blast

from \( c \) show \( \text{False} \)

by blast

qed
Isar_Demo.thy

Cantor and abbreviations
**Abbreviations**

*this* = the previous proposition proved or assumed

*then* = from *this*

*thus* = then show

*hence* = then have
using and with

\[(\text{have|show}) \text{ prop using facts} = \text{from facts (have|show) prop with facts} = \text{from facts this}\]
lemma

fixes $f :: 'a \Rightarrow 'a \text{ set}$
assumes $s : \text{surj } f$
shows $\text{False}$

proof — no automatic proof step

have $\exists \ a. \ \{ x. \ x \notin f \ x \} = f \ a$ using $s$
  by (auto simp: surj_def)

thus $\text{False}$ by blast

qed

Proves $\text{surj } f \implies \text{False}$

but $\text{surj } f$ becomes local fact $s$ in proof.
The essence of structured proofs

Assumptions and intermediate facts can be named and referred to explicitly and selectively
Structured lemma statements

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

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Case distinction

show \( R \)
proof cases
  assume \( P \)
  :
  show \( R \) ...
next
  assume \( \neg P \)
  :
  show \( R \) ...
qed

have \( P \lor Q \) ...
then show \( R \)
proof
  assume \( P \)
  :
  show \( R \) ...
next
  assume \( Q \)
  :
  show \( R \) ...
qed
show \neg P
proof
  assume P
  
  show False ...
qed

show P
proof (rule ccontr)
  assume \neg P
  
  show False ...
qed
show $P \iff Q$

proof

assume $P$

\vdots

show $Q$ \ldots

next

assume $Q$

\vdots

show $P$ \ldots

qed
∀ and ∃ introduction

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

show \( \exists x. \ P(x) \)
proof
  :  :
  show \( P(\text{witness}) \)  ...
qed
\( \exists \text{ elimination: obtain} \)

have \( \exists x. \ P(x) \)
then obtain \( x \) where \( p: \ P(x) \) by blast
\[ x \text{ fixed local variable} \]

Works for one or more \( x \)
lemma \rightarrow \text{surj}(f :: \ 'a \Rightarrow \ 'a \text{ set})
proof
  assume \text{surj } f
  hence \exists a. \{x. x \notin f \ x\} = f \ a \ by (auto simp: \text{surj_def})
  then obtain a where \{x. x \notin f \ x\} = f \ a \ by \text{ blast}
  hence a \notin f \ a \leftrightarrow a \in f \ a \ by \text{ blast}
  thus \text{False} \ by \text{ blast}
qed
Set equality and subset

show \( A = B \)

proof

show \( A \subseteq B \) \ldots

next

show \( B \subseteq A \) \ldots

qed

show \( A \subseteq B \)

proof

fix \( x \)

assume \( x \in A \)

: \quad show \( x \in B \) \ldots

qed
Isar_Demo.thy

Exercise
12 Isar by example
13 Proof patterns
14 Streamlining Proofs
15 Proof by Cases and Induction
Streamlining Proofs
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Example: pattern matching

\[
\text{show } \text{formula}_1 \leftrightarrow \text{formula}_2 \quad (\text{is } \text{?L} \leftrightarrow \text{?R})
\]

\text{proof}

\begin{align*}
\text{assume } \text{?L} \\
\vdots \\
\text{show } \text{?R} \ldots \\
\end{align*}

\text{next}

\begin{align*}
\text{assume } \text{?R} \\
\vdots \\
\text{show } \text{?L} \ldots \\
\end{align*}

\text{qed}
show \textit{formula} \ (is \ ?thesis)

proof -
 :

  show \ ?thesis \ 

qed

Every show implicitly defines \ ?thesis
Introducing local abbreviations in proofs:

```ml
let ?t = "some-big-term"

have "... ?t ..."
```
Quoting facts by value

By name:

```latex
\textbf{have } x0: "x > 0" \ldots \\
\vdots \\
\textbf{from } x0 \ldots
```

By value:

```latex
\textbf{have } "x > 0" \ldots \\
\vdots \\
\textbf{from } 'x>0' \ldots
```

↑ ↑

back quotes
Isar_Demo.thy

Pattern matching and quotation
14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks
Example

lemma

\( (\exists \ ys \ zs. \ xs = ys @ zs \land \text{length } ys = \text{length } zs) \lor \\
(\exists \ ys \ zs. \ xs = ys @ zs \land \text{length } ys = \text{length } zs + 1) \)

proof ???
Isar_Demo.thy

Top down proof development
When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

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

At the end:

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

Pattern Matching and Quotations

Top down proof development

*moreover* and *raw* proof blocks
moreover—ultimately

\[ \text{have } P_1 \ldots \text{ have } P_2 \ldots \text{ have } P_n \ldots \text{ ultimately have } P \ldots \]

\[ \text{have } \text{lab}_1: P_1 \ldots \text{ have } \text{lab}_2: P_2 \ldots \vdots \text{ have } \text{lab}_n: P_n \ldots \text{ from } \text{lab}_1 \text{ lab}_2 \ldots \text{ have } P \ldots \]

With names
\[
\begin{aligned}
\{ & \textbf{fix} \ x_1 \ldots x_n \\
& \textbf{assume} \ A_1 \ldots A_m \\
& \vdots \\
& \textbf{have} \ B
\}
\end{aligned}
\]

proves \[
\left[ A_1; \ldots; A_m \right] \implies B
\]
where all \( x_i \) have been replaced by \(?x_i\).
Isar_Demo.thy

moreover and {   }
Proof state and Isar text

In general: \textbf{proof method}

Applies \textit{method} and generates subgoal(s):
\[
\bigwedge x_1 \ldots x_n \left[ A_1; \ldots ; A_m \right] \Longrightarrow B
\]

How to prove each subgoal:

\begin{verbatim}
fix \( x_1 \ldots x_n \)
assume \( A_1 \ldots A_m \)
:
show \( B \)
\end{verbatim}

Separated by \textbf{next}
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Isar_Induction_Demo.thy

Proof by cases
Datatype case analysis

datatype \( t = C_1 \vec{\tau} | \ldots \)

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

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

fix \(x_1 \ldots x_k\)
assume \(C_i: \) \(\begin{cases} \text{label} \quad \text{term} = (C_i x_1 \ldots x_k) \end{cases}\)
Isar_Induction_Demo.thy

Structural induction for $\textit{nat}$
Structural induction for \textit{nat}

\textbf{show} \quad P(n)
\begin{align*}
\textbf{proof} \quad (\text{induction } n) \\
\quad \text{case} \ 0 & \quad \equiv \quad \text{let} \ \ ?\text{case} = P(0) \\
\quad \vdots & \quad \equiv \quad \text{fix} \ n \ \text{assume} \ Suc: \ P(n) \\
\quad \text{show} \ ?\text{case} & \quad \equiv \quad \text{let} \ ?\text{case} = P(Suc \ n) \\
\textbf{next} & \quad \vdots \\
\quad \text{case} \ (Suc \ n) & \\
\quad \vdots & \\
\quad \text{show} \ ?\text{case} & \\
\textbf{qed}
\end{align*}
Structural induction with $\Rightarrow$

$\text{show } A(n) \implies P(n)$

$\text{proof (induction } n)$

$\text{case } 0$

$\text{let } ?\text{case }= P(0)$

$\text{show } ?\text{case}$

$\text{next}$

$\text{case } (Suc \ n)$

$\text{assume } Suc: A(n) \implies P(n)$

$\text{let } ?\text{case }= P(Suc \ n)$

$\text{show } ?\text{case}$

qed
Named assumptions

In a proof of

\[ A_1 \implies \ldots \implies A_n \implies B \]

by structural induction:

In the context of

\text{case } C

we have

\begin{align*}
\text{C.IH} & \quad \text{the induction hypotheses} \\
\text{C.prem} & \quad \text{the premises } A_i \\
\text{C} & \quad \text{C.IH } + \text{ C.prem} \\
\end{align*}
A remark on style

- **case** \((\text{Suc } n) \ldots \text{show } ?\text{case}\)
  is easy to write and maintain

- **fix** \(n\) **assume** \(\text{formula} \ldots \text{show } \text{formula'}\)
  is easier to read:
  - all information is shown locally
  - no contextual references (e.g. \(?\text{case}\)
Proof by Cases and Induction

Rule Induction

Rule Inversion
Isar_Induction_Demo.thy

Rule induction
Rule induction

\textbf{inductive} \( I :: \tau \Rightarrow \sigma \Rightarrow \text{bool} \) where
\begin{align*}
\text{rule}_1: & \ldots \\
\vdots \\
\text{rule}_n: & \ldots
\end{align*}

\textbf{show} \( I \ x \ y \Rightarrow P \ x \ y \)
\textbf{proof} \ (\text{induction rule: } I\.\text{induct})
\begin{align*}
\text{case} & \ \text{rule}_1 \\
\vdots \\
\text{show} & \ ?\text{case} \\
\text{next} \\
\vdots \\
\text{next} & \ \text{rule}_n \\
\vdots \\
\text{show} & \ ?\text{case}
\end{align*}
\textbf{qed}
Fixing your own variable names

\textbf{case } (\textit{rule}_i \ x_1 \ldots \ x_k)

Renames the first $k$ variables in $\textit{rule}_i$ (from left to right) to $x_1 \ldots \ x_k$. 
Named assumptions

In a proof of

\[ I \ldots \implies A_1 \implies \ldots \implies A_n \implies B \]

by rule induction on \( I \ldots : \)

In the context of

**case** \( R \)

we have

- \( R.IH \) the induction hypotheses
- \( R.hyps \) the assumptions of rule \( R \)
- \( R.prems \) the premises \( A_i \)

\[ R \; R.IH + R.hyps + R.prems \]
**Rule inversion**

**inductive** \( ev :: \text{nat} \Rightarrow \text{bool} \) where  
- \( ev0 : ev \ 0 \ |
- \( evSS : ev \ n \implies ev(Suc(Suc\ n)) \)

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

\[
ev \ n \implies n = 0 \lor (\exists k. \ n = Suc(Suc \ k) \land ev \ k)
\]

**Rule inversion = case distinction over rules**
Isar_Induction_Demo.thy

Rule inversion
Rule inversion template

from 'ev n' have P

proof cases
  case ev0
  : show ?thesis ...

next
  case (evSS k)
  : show ?thesis ...

qed

Impossible cases disappear automatically
Part II

Semantics
Chapter 7

IMP:
A Simple Imperative Language
IMP Commands

Big-Step Semantics

Small-Step Semantics
16 IMP Commands

17 Big-Step Semantics

18 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:

\[
\text{com} ::= \text{SKIP} \\
| \text{string} ::= \text{aexp} \\
| \text{com} ;; \text{com} \\
| \text{IF} \ bexp \ \text{THEN} \ \text{com} \ \text{ELSE} \ \text{com} \\
| \text{WHILE} \ bexp \ \text{DO} \ \text{com}
\]
Commands

Abstract syntax:

\[
\textbf{datatype} \ com \ = \ \texttt{SKIP} \\
\texttt{Assign \ string \ aexp} \\
\texttt{Seq \ com \ com} \\
\texttt{If \ bexp \ com \ com} \\
\texttt{While \ bexp \ com}
\]
Com.thy
IMP Commands

Big-Step Semantics

Small-Step Semantics
Big-step semantics

Concrete syntax:

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

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

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

\[(\text{SKIP}, s) \Rightarrow s\]

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

\[
\frac{(c_1, s_1) \Rightarrow s_2 \quad (c_2, s_2) \Rightarrow s_3}{(c_1;; c_2, s_1) \Rightarrow s_3}
\]
Big-step rules

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

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

\[
\frac{(c, \ s_1) \Rightarrow \ s_2 \quad bval \ b \ s_1}{(WHILE \ b \ DO \ c, \ s_2) \Rightarrow \ s_3}
\]

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

\[
\begin{align*}
(\"x\" &: N 5; \"y\" &: V \"x\", s) & \Rightarrow ? \\
(w, s_i) & \Rightarrow ?
\end{align*}
\]

where

\[
\begin{align*}
w &= \text{WHILE } b \text{ DO } c \\
b &= \text{NotEq } (V \"x\") (N 2) \\
c &= \"x\" &: \text{Plus } (V \"x\") (N 1) \\
s_i &= s(\"x\" := i) \\
\text{NotEq } a_1 \ a_2 &= \\
\text{Not(And (Not(Less a_1 \ a_2)) (Not(Less a_2 \ a_1))})
\end{align*}
\]
Logically speaking

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

is just infix syntax for

\[\text{big_step } (c,s) \; t\]

where

\[\text{big_step :: com } \times \text{ state } \Rightarrow \text{ state } \Rightarrow \text{ bool}\]

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 `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

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

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*

\[
\begin{array}{c}
asm 
asm_1 \implies P 
\vdots 
asm_n \implies P \\
\hline
P
\end{array}
\]

(possibly with \(\bigwedge x\) in front of the \(asm_i \implies P\))

Reading:

To prove a goal \(P\) with assumption \(asm\), prove all \(asm_i \implies P\)

Example:

\[
\begin{array}{c}
F \lor G 
F \implies P 
G \implies P \\
\hline
P
\end{array}
\]
Theorems with `elim` attribute are used automatically by `blast`, `fastforce` and `auto`.

- Can also be added locally, e.g., `(blast elim: ...)`

- Variant: `elim!` applies elim-rules eagerly.
Big_Step.thy

Rule inversion
Command equivalence

Two commands have the same input/output behaviour:

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

Example

\[ w \sim w' \]

where  \[ w = WHILE \ b \ DO \ c \]
\[ w' = IF \ b \ THEN \ c;; \ w \ ELSE \ SKIP \]
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:

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

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

We need a finer grained semantics!
IMP Commands

Big-Step Semantics

Small-Step Semantics
Small-step semantics

Concrete syntax:

\[(\textit{com}, \textit{state}) \rightarrow (\textit{com}, \textit{state})\]

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 \(\neg (\exists cs'. \ cs \rightarrow cs')\).
The intention:

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

Why?

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

\[(x ::= a, s) \rightarrow (\text{SKIP}, s(x ::= \text{aval } a \ s))\]

\[(\text{SKIP};; c, s) \rightarrow (c, s)\]

\[\frac{(c_1, s) \rightarrow (c'_1, s')}{(c_1; ; c_2, s) \rightarrow (c'_1; ; c_2, s')}\]
Small-step rules

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

Fact \((\text{SKIP}, s)\) is a final configuration.
Small-step examples

\[
\left( "z" ::= V "x"\!,\; "x" ::= V "y"\!,\; "y" ::= V "z",\; s \right) \rightarrow \\
\ldots
\]

where \( s = \langle "x" := 3,\; "y" := 7,\; "z" := 5 \rangle \).

\[
\left( w,\; s_0 \right) \rightarrow \ldots
\]

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

\[
\begin{align*}
b &= \text{Less}\; (V "x")\; (N\; 1) \\
c &= "x" ::= \text{Plus}\; (V "x")\; (N\; 1) \\
s_n &= \langle "x" := n \rangle
\end{align*}
\]
Are big and small-step semantics equivalent?
Theorem \( cs \Rightarrow t \implies cs \rightarrow^* (SKIP, t) \)

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

In two cases a lemma is needed:

**Lemma**

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

Proof by rule induction.
Theorem \( cs \rightarrow^* (SKIP, t) \implies cs \Rightarrow t \)

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

In the induction step a lemma is needed:

**Lemma** \( cs \rightarrow cs' \implies cs' \Rightarrow t \implies cs \Rightarrow t \)

Proof by rule induction on \( cs \rightarrow cs' \).
Corollary $cs \Rightarrow t \leftrightarrow cs \rightarrow^* (SKIP, t)$
Small_Step.thy

Equivalence of big and small
Can execution stop prematurely?

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

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

We prove the contrapositive

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

by induction on \(c\).

- **Case** \(c_1 ;; c_2\): by case distinction:
  - \(c_1 = SKIP \implies \neg \text{final} (c_1 ;; c_2, s)\)
  - \(c_1 \neq SKIP \implies \neg \text{final} (c_1, s)\) (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  → terminates

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

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

=  \((\exists t. \ cs \rightarrow^* (\text{SKIP}, t))\)

(by big = small)

=  \((\exists \ cs'. \ cs \rightarrow^* \ cs' \land \text{final} \ cs')\)

(by final = SKIP)

Equivalent:

⇒ does not yield final state  iff  → 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 \( \rightarrow \) path)
- **must** terminate (all \( \rightarrow \) paths terminate)

Therefore: \( \Rightarrow \) correctly reflects termination behaviour.

With nondeterminism: may have both \( cs \Rightarrow t \) and a nonterminating reduction \( cs \rightarrow cs' \rightarrow \ldots \)
Chapter 8

Compiler
Stack Machine

Compiler
Stack Machine

Compiler
## Stack Machine

**Instructions:**

```plaintext
datatype instr =
   LOADI int          load value
   LOAD vname         load var
   ADD                add top of stack
   STORE vname        store var
   JMP int            jump
   JMPLESS int        jump if <
   JMPGE int          jump if ≥
```

Type synonyms:

\[\text{stack} = \text{int} \ \text{list}\]
\[\text{config} = \text{int} \ \times \ \text{state} \ \times \ \text{stack}\]

Execution of 1 instruction:

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

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

Programs are instruction lists.

Executing one program step:

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

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

where \( 'a \ \text{list} !! \text{int} \) = nth instruction of list
and \( \text{size :: list } \Rightarrow \text{int} \) = list size as integer
Program execution (∗ steps)

Defined in the usual manner:

\[ P \vdash (pc, s, stk) \rightarrow^* (pc', s', stk') \]
Compiler.thy

Stack Machine
19 Stack Machine

20 Compiler
Compiling \textit{aexp}

Same as before:

\[
\begin{align*}
\text{acomp}\ (N\ n) &= [\text{LOADI}\ n] \\
\text{acomp}\ (V\ x) &= [\text{LOAD}\ x] \\
\text{acomp}\ (\text{Plus}\ a_1\ a_2) &= \text{acomp}\ a_1 \oplus \text{acomp}\ a_2 \oplus [\text{ADD}]
\end{align*}
\]

Correctness theorem:

\[
\text{acomp}\ a \vdash (\emptyset, s, stk) \rightarrow^* (\text{size}\ (\text{acomp}\ a), s, \text{aval}\ a\ s \neq stk)
\]

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

Needs lemmas!
\[ P \vdash c \rightarrow^* c' \quad \Rightarrow \quad P @ P' \vdash c \rightarrow^* c' \]

\[ P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \quad \Rightarrow \quad P' @ P \vdash (size P' + i, s, stk) \rightarrow^* (size P' + i', s', stk') \]

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

\[ P \vdash c \rightarrow c' \quad \Rightarrow \quad P @ P' \vdash c \rightarrow c' \]

\[ P \vdash (i, s, stk) \rightarrow (i', s', stk') \quad \Rightarrow \quad P' @ P \vdash (size P' + i, s, stk) \rightarrow (size P' + i', s', stk') \]

Proofs by cases.
Compiling $bexp$

Let $ins$ be the compilation of $b$:

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

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

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

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

Let $b = \text{And} \ (\text{Less} \ (V "x") \ (V "y")) \ (\text{Not} \ (\text{Less} \ (V "z") \ (V "a")))$.

$b\text{comp} b \ False \ 3 = [\text{LOAD} "x", \text{LOAD} "y", \text{LOAD} "z", \text{LOAD} "a", ]$
\[ bcomp : bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list \]

\[ bcomp (Bc\ v) f n = (if\ v = f\ then\ [JMP\ n]\ else\ []) \]

\[ bcomp (Not\ b) f n = bcomp\ b\ (\neg f)\ n \]

\[ bcomp (Less\ a_1\ a_2) f n = \]

\[ acomp\ a_1@ \]
\[ acomp\ a_2@ (if\ f\ then\ [JMPLESS\ n]\ else\ [JMPGE\ n]) \]

\[ bcomp (And\ b_1\ b_2) f n = \]

\[ let\ cb_2 = bcomp\ b_2\ f\ n;\]
\[ m = if\ f\ then\ size\ cb_2\ else\ size\ cb_2 + n;\]
\[ cb_1 = bcomp\ b_1\ False\ m \]
\[ in\ cb_1@ cb_2 \]
\[ 0 \leq n \implies \]  
\[ \text{bcomp } b \ f \ n \]  
\[ \vdash (0, s, stk) \rightarrow^* \]  
\[ (\text{size } (\text{bcomp } b \ f \ n) + (\text{if } f = \text{bval } b \ s \ \text{then } n \ \text{else } 0), s, stk) \]
Compiling \textit{com}

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

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

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

\[
\text{ccomp}\ (c_1;; c_2) = \text{ccomp}\ c_1 \odot \text{ccomp}\ c_2
\]
\[ ccomp \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) = \]

\[
let \ cc_1 = ccomp \ c_1; \ cc_2 = ccomp \ c_2; \\
    cb = bcomp \ b \ False \ (size \ cc_1 + 1) \\
in \ cb \ @ \ cc_1 \ @ \ JMP \ (size \ cc_2) \ # \ cc_2
\]
\[ \text{ccomp } (\text{WHILE } b \text{ DO } c) = \]

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

- code for \( b \)
- code for \( c \)
Correctness of $ccomp$

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

$$(c, s) \implies t \implies ccomp \vdash (0, s, stk) \rightarrow^* (\text{size} (ccomp c), t, stk)$$

Proof by rule induction.
The other direction

We have only shown “$\implies$”:

*compiled code simulates source code.*

How about “$\impliedby$”:

*source code simulates compiled code?*

If $c_\text{comp} \ c$ with start state $s$ produces result $t$, and if(!) $(c, s) \implies t'$, then “$\implies$” implies that $c_\text{comp} \ c$ 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 $c_\text{comp} \ c$ does.
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
A Typed Version of IMP

Security Type Systems
A Typed Version of IMP

Security Type Systems
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
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 \textit{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:

**datatype val** = \textit{Iv} int \ | \ \textit{Rv} real

The state:

\textit{state} = \textit{vname} \Rightarrow \textit{val}

Arithmetic expressions:

**datatype aexp** =

\textit{Ic} int \ | \ \textit{Rc} real \ | \ \textit{V} vname \ | \ \textit{Plus aexp aexp}
Why tagged values?

Because we want to detect if things “go wrong”.
What can go wrong? Adding integer and real!
No automatic coercions.
Does this mean any implementation of IMP also needs to tag values?
No! Compilers compile only well-typed programs, and well-typed programs do not need tags.

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

Not recursive function but inductive predicate:

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

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

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

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

$taval \ a_1 \ s \ (Iv \ i_1) \quad taval \ a_2 \ s \ (Iv \ i_2)$

$taval (Plus \ a_1 \ a_2) \ s \ (Iv \ (i_1 + i_2))$

$taval \ a_1 \ s \ (Rv \ r_1) \quad taval \ a_2 \ s \ (Rv \ r_2)$

$taval (Plus \ a_1 \ a_2) \ s \ (Rv \ (r_1 + r_2))$
Example: evaluation of $\texttt{Plus} \ (V "x") \ (Ic \ 1)$

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

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

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

If $s "x" = Rv \ r$: then there is \textit{no} value $v$ such that $\text{taval} \ (\texttt{Plus} \ (V "x") \ (Ic \ 1)) \ s \ v$. 
The functional alternative

taval :: \textit{aexp} \rightarrow \textit{state} \rightarrow \textit{val} \textit{option}

Exercise!
Boolean expressions

Syntax as before. Semantics:

\[
\begin{align*}
\text{tbval} &: \text{bexp} \Rightarrow \text{state} \Rightarrow \text{bool} \Rightarrow \text{bool} \\
\text{tbval} (Bc \; v) \; s \; v & \quad \frac{\text{tbval} \; b \; s \; \text{bv}}{\text{tbval} (\text{Not} \; b) \; s \; (\neg \; \text{bv})} \\
\text{tbval} \; b_1 \; s \; \text{bv}_1 & \quad \text{tbval} \; b_2 \; s \; \text{bv}_2 \\
\text{tbval} (\text{And} \; b_1 \; b_2) \; s \; (\text{bv}_1 \land \text{bv}_2) & \\
\text{taval} \; a_1 \; s \; (\text{Iv} \; i_1) & \quad \text{taval} \; a_2 \; s \; (\text{Iv} \; i_2) \\
\text{tbval} (\text{Less} \; a_1 \; a_2) \; s \; (i_1 < i_2) & \\
\text{taval} \; a_1 \; s \; (\text{Rv} \; r_1) & \quad \text{taval} \; a_2 \; s \; (\text{Rv} \; r_2) \\
\text{tbval} (\text{Less} \; a_1 \; a_2) \; s \; (r_1 < r_2) 
\end{align*}
\]
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

\[
\text{taval} \ a \ s \ v \\
(x ::= a, s) \rightarrow (\text{SKIP}, s(x ::= v))
\]

\[
\text{tbval} \ b \ s \ \text{True} \\
(IF \ b \ \text{THEN} \ c_1 \ ELSE \ c_2, s) \rightarrow (c_1, s)
\]

\[
\text{tbval} \ b \ s \ \text{False} \\
(IF \ b \ \text{THEN} \ c_1 \ ELSE \ c_2, s) \rightarrow (c_2, s)
\]

The other rules remain unchanged.
Example

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

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

- If $s \; "x" = Rv \; r$:
  $$ (c, \; s) \nRightarrow $$
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 $\text{Plus} \ (V "x") \ (V "y")$?

Depends on the type of "$x" and "$y"!

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

$\text{tyenv} = vname \Rightarrow ty$

The type of an expression is always relative to a type enviroment $\Gamma$. Standard notation:

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

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

$\Gamma \vdash a : \tau$

$tyenv \vdash aexp : ty$

The rules:

$\Gamma \vdash Ic\ i : Ity$

$\Gamma \vdash Rc\ r : Rty$

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

$\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau$

Therefore

$\Gamma \vdash Plus\ a_1\ a_2 : \tau$
Example

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

where $\Gamma \ "x" = Ity$. 
Notation:

\[ \Gamma \vdash b \]

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

Read: \textit{In context }\Gamma, \textit{ }b \textit{ is well-typed.}
The rules:

\[ \Gamma \vdash Bc \; v \]
\[ \Gamma \vdash b \]
\[ \Gamma \vdash \text{Not } b \]

\[ \begin{array}{c}
\Gamma \vdash b_1 \\
\Gamma \vdash b_2
\end{array} \]
\[ \Gamma \vdash \text{And } b_1 \; b_2 \]

\[ \begin{array}{c}
\Gamma \vdash a_1 : \tau \\
\Gamma \vdash a_2 : \tau
\end{array} \]
\[ \Gamma \vdash \text{Less } a_1 \; a_2 \]

Example:  \( \Gamma \vdash \text{Less } (Ic \; i) \; (Rc \; r) \) does not hold.
Well-typed commands

Notation:

\[ \Gamma \vdash c \]

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

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

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

\[
\Gamma \vdash a : \Gamma \quad x \\
\Gamma \vdash x ::= a
\]

\[
\Gamma \vdash c_1 \quad \Gamma \vdash c_2 \\
\quad \quad \quad \quad \Gamma \vdash c_1 ;; c_2
\]

\[
\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2 \\
\quad \quad \quad \quad \Gamma \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2
\]

\[
\Gamma \vdash b \quad \Gamma \vdash c \\
\quad \quad \quad \quad \Gamma \vdash \text{WHILE } b \ \text{DO } c
\]
Syntax-directedness

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

- There is exactly one rule for each syntactic construct ($\texttt{SKIP}$, ::=, ...).
- 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 \texttt{WHILE} depends on the execution of \texttt{WHILE}.
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 \( (x := \text{Plus} (V \, x) (Ic 1), s) \not\rightarrow \)

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

\[ \text{type } (Iv ~ i) = Ity \]
\[ \text{type } (Rv ~ r) = Rty \]

Well-typed state:

\[ \Gamma \vdash s \leftrightarrow (\forall x. \text{type } (s ~ x) = \Gamma ~ x) \]
Type soundness

Reduction cannot get stuck:

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

Follows from progress:

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

and preservation:

If everything is ok and you take a step, then everything is ok again.
The slogan

\[ \text{Progress} \land \text{Preservation} \implies \text{Type safety} \]

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

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

Preservation: Well-typedness is an *invariant*. 

Progress:

\[ \Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP} \] \implies \exists cs'. (c, s) \to cs'

Preservation:

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

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

Type soundness:

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

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

\[ \Gamma \vdash a : \tau; \Gamma \vdash s \] \quad \Rightarrow \quad \exists v. \text{taval } a \ s \ v

Preservation:

\[ \Gamma \vdash a : \tau; \text{taval } a \ s \ v; \Gamma \vdash s \] \quad \Rightarrow \quad \text{type } v = \tau
All proofs by rule induction.
Types.thy
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 known as *information flow control*. 
Language based security is an approach to information flow control where data flow analysis is used to determine whether a program is free of illicit information flows.

LBS guarantees confidentiality by program analysis, not by cryptography.

These analyses are often expressed as type systems.
Security levels

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

Explicit: \( \text{low} := \text{high} \)

Implicit: if \( \text{high1} = \text{high2} \) then \( \text{low} := 1 \)
else \( \text{low} := 0 \)
Noninterference

High variables do not interfere with low ones.

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

Program $c$ guarantees noninterference iff for all $s_1, s_2$:

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

Secure IMP

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

Security levels:

**type_synonym**  \( level = \text{nat} \)

Every variable has a security level:

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

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

\[ \text{sec} \ x = \text{length} \ x \]
The security level of an expression is the maximal security level of any of its variables.

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

\[ \text{sec} (N \ n) = 0 \]
\[ \text{sec} (V \ x) = \text{sec} \ x \]
\[ \text{sec} (\text{Plus} \ a \ b) = \text{max} (\text{sec} \ a) (\text{sec} \ b) \]
Security Levels on $bexp$

$sec :: bexp \Rightarrow level$

$sec \ (Bc \ v) = 0$

$sec \ (Not \ b) = sec \ b$

$sec \ (And \ b_1 \ b_2) = max \ (sec \ b_1) \ (sec \ b_2)$

$sec \ (Less \ a \ b) = max \ (sec \ a) \ (sec \ b)$
Security Levels on States

Agreement of states up to a certain level:

\[ s_1 = s_2 \ (\leq l) \iff \forall x. \ sec x \leq l \rightarrow s_1 \ x = s_2 \ x \]

\[ s_1 = s_2 \ (< l) \iff \forall x. \ sec x < l \rightarrow s_1 \ x = s_2 \ x \]

Noninterference lemmas for expressions:

\[
\begin{align*}
\frac{s_1 = s_2 \ (\leq l) \quad sec \ a \leq l}{aval \ a \ s_1 = aval \ a \ s_2} \\
\frac{s_1 = s_2 \ (\leq l) \quad sec \ b \leq l}{bval \ b \ s_1 = bval \ b \ s_2}
\end{align*}
\]
Security Type Systems
Secure IMP
A Security Type System
A Type System with Subsumption
A Bottom-Up Type System
Beyond
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 $c = \begin{cases} \text{IF } \text{Less} \ (V "x1") \ (V "x") \ \text{THEN } "x1" ::= N \ 0 \\
\text{ELSE } "x1" ::= N \ 1 \end{cases}$

1 $\vdash c$ ? Yes

2 $\vdash c$ ? Yes

3 $\vdash c$ ? No
The type system

\[ l \vdash SKIP \]

\[
\frac{sec a \leq sec x \quad l \leq sec x}{l \vdash x ::= a}
\]

\[
\frac{l \vdash c_1 \quad l \vdash c_2}{l \vdash c_1 ;; c_2}
\]

\[
\frac{\max (sec b) \quad l \vdash c_1 \quad \max (sec b) \quad l \vdash c_2}{l \vdash IF b THEN c_1 ELSE c_2}
\]

\[
\frac{\max (sec b) \quad l \vdash c}{l \vdash WHILE b DO c}
\]
Remark:

\[ l \vdash c \text{ is syntax-directed and executable.} \]
Anti-monotonicity

Proof by ... as usual.

This is often called a subsumption rule because it says that larger levels subsume smaller ones.
If $l \vdash c$ then $c$ cannot modify variables of level $< l$:

\[
\begin{align*}
(c, s) &\Rightarrow t \\ l &\vdash c \\
\hline
s &= t (< l)
\end{align*}
\]

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

Proof by \ldots as usual.
Noninterference

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

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

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:
\[
l \vdash ' \text{SKIP}
\]
\[
\text{sec } a \leq \text{sec } x \quad l \leq \text{sec } x
\]
\[
l \vdash ' x ::= a
\]
\[
l \vdash ' c_1 \quad l \vdash ' c_2
\]
\[
l \vdash ' c_1 ;; c_2
\]
\[
\text{sec } b \leq l \quad l \vdash ' c_1 \quad l \vdash ' c_2
\]
\[
l \vdash ' \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2
\]
\[
\text{sec } b \leq l \quad l \vdash ' c
\]
\[
l \vdash ' \text{WHILE } b \ \text{DO } c
\]
The subsumption-based system $\vdash'$ is neither syntax-directed nor directly executable.

Need to guess when to use the subsumption rule.
Equivalence of $\vdash$ and $\vdash'$

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

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

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

Proof by induction. Subsumption already a lemma for $\vdash$. 
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.
\[ \vdash \text{SKIP} : l \]

\[ \text{sec } a \leq \text{sec } x \]
\[ \vdash x ::= a : \text{sec } x \]

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

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

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

\[ \vdash c : l \iff l \vdash' c \]

Proof by induction.

\[ l \vdash' c \iff \vdash c : l \]

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

\[ l \vdash' c \iff \exists l' \geq l. \vdash c : l' \]

Proof by induction.
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?

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

Beware of covert channels!

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

\[ \textit{WHILE}-\text{conditions must not depend on confidential data.} \]

New typing rule:

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

Now provable:

\[
\begin{align*}
(c, s) \Rightarrow s' & \quad 0 \vdash c \quad s = t \ (\leq l) \\
\exists t'. (c, t) \Rightarrow t' & \land s' = t' \ (\leq l)
\end{align*}
\]
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
Definite Initialization Analysis

Live Variable Analysis
Definite Initialization Analysis

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:

\[
\text{IF } x < 1 \text{ THEN } y := x \text{ ELSE } y := x + 1; \\
y := y + 1
\]

\[
\text{IF } x < x \text{ THEN } y := y + 1 \text{ ELSE } y := x
\]

Assume $x$ and $y$ are initialized:

\[
\text{WHILE } x < y \text{ DO } z := x; \ z := z + 1
\]
Simplifying principle

*We do not analyze boolean expressions to determine program execution.*
23 Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics
Theory \emph{Vars} provides an overloaded function \emph{vars}:

\begin{align*}
\text{vars} &:: \text{aexp} \Rightarrow \text{vname set} \\
\text{vars} (N\ n) & = \{\} \\
\text{vars} (V\ x) & = \{x\} \\
\text{vars} (\text{Plus}\ a_1\ a_2) & = \text{vars} a_1 \cup \text{vars} a_2 \\
\text{vars} &:: \text{bexp} \Rightarrow \text{vname set} \\
\text{vars} (Bc\ v) & = \{\} \\
\text{vars} (\text{Not}\ b) & = \text{vars} b \\
\text{vars} (\text{And}\ b_1\ b_2) & = \text{vars} b_1 \cup \text{vars} b_2 \\
\text{vars} (\text{Less}\ a_1\ a_2) & = \text{vars} a_1 \cup \text{vars} a_2
\end{align*}
Vars.thy
23  Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics
Modified example from the JLS:

Variable $x$ is definitely initialized after $SKIP$

iff $x$ is definitely initialized before $SKIP$.

Similar statements for each language construct.
$\vdash \text{vname set } \Rightarrow \text{com } \Rightarrow \text{vname set } \Rightarrow \text{bool}$

$D \ A \ c \ A'$ 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.
\[ D \ A \ SKIP \ A \]

\[
\text{vars } a \subseteq A
\]

\[ D \ A \ (x ::= a) \ (\text{insert } x \ A) \]

\[
D \ A_1 \ c_1 \ A_2 \quad D \ A_2 \ c_2 \ A_3
\]

\[ D \ A_1 \ (c_1 ; ; c_2) \ A_3 \]

\[
\text{vars } b \subseteq A \quad D \ A \ c_1 \ A_1 \quad D \ A \ c_2 \ A_2
\]

\[ D \ A \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ (A_1 \ \cap \ A_2) \]

\[
\text{vars } b \subseteq A \quad D \ A \ c \ A' \]

\[ D \ A \ (WHILE \ b \ DO \ c) \ A \]
Correctness of $D$

- Things can go wrong: execution may access uninitialized variable.
  \[\Rightarrow\] 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
\[ \text{state} = \text{vname} \Rightarrow \text{val option} \]

where

**datatype** \( 'a \text{ option} = \text{None} | \text{Some } 'a \)

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

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

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

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

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

\[
\text{aval (Plus a}_1 \ a_2) s =
\]

\[
\text{(case } \text{aval a}_1 \ s, \text{ aval a}_2 \ s) \text{ of}
\]

\[
(\text{Some i}_1, \text{ Some i}_2) \Rightarrow \text{Some(i}_1 + i_2)
\]

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

\text{bval} (Bc \ v) \ s = \text{Some} \ v

\text{bval} (\text{Not} \ b) \ s =
(\text{case} \ \text{bval} \ b \ s \ \text{of} \ None \Rightarrow \text{None}
| \ \text{Some} \ bv \Rightarrow \text{Some} \ (\neg \ bv))

\text{bval} (\text{And} \ b_1 \ b_2) \ s =
(\text{case} \ (\text{bval} \ b_1 \ s, \ \text{bval} \ b_2 \ s) \ \text{of}
  (\text{Some} \ bv_1, \ \text{Some} \ bv_2) \Rightarrow \text{Some}(bv_1 \land bv_2)
| \ _ \Rightarrow \text{None})

\text{bval} (\text{Less} \ a_1 \ a_2) \ 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})
Big step semantics

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

A small complication:

\[
\begin{align*}
 (c_1, s_1) &\Rightarrow \text{Some } s_2 & (c_2, s_2) &\Rightarrow s \\
 (c_1 ;; c_2, s_1) &\Rightarrow s \\
 (c_1, s_1) &\Rightarrow \text{None} \\
 (c_1 ;; c_2, s_1) &\Rightarrow \text{None}
\end{align*}
\]

More convenient, because compositional:

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

\[(c, \text{None}) \Rightarrow \text{None}\]

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

\[(\text{SKIP}, s) \Rightarrow s\]

\[\text{aval } a \ s = \text{Some } i\]

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

\[\text{aval } a \ s = \text{None}\]

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

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

\[(c_1 ;; c_2, s_1) \Rightarrow s_3\]
\[
\begin{align*}
\text{bval } b \ s &= \text{Some True} \quad (c_1, \text{Some } s) \Rightarrow s' \\
(IF \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, \ \text{Some } s) &\Rightarrow s' \\
\text{bval } b \ s &= \text{Some False} \quad (c_2, \text{Some } s) \Rightarrow s' \\
(IF \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, \ \text{Some } s) &\Rightarrow s' \\
\text{bval } b \ s &= \text{None} \\
(IF \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, \ \text{Some } s) &\Rightarrow \text{None}
\end{align*}
\]
\[ \text{bval } b \ s = \text{Some False} \]
\[ (\text{WHILE } b \text{ DO } c, \text{Some } s) \Rightarrow \text{Some } s \]

\[ \text{bval } b \ s = \text{Some True} \]
\[ (c, \text{Some } s) \Rightarrow s' \quad (\text{WHILE } b \text{ DO } c, s') \Rightarrow s'' \]
\[ (\text{WHILE } b \text{ DO } c, \text{Some } s) \Rightarrow s'' \]

\[ \text{bval } b \ s = \text{None} \]
\[ (\text{WHILE } b \text{ DO } c, \text{Some } s) \Rightarrow \text{None} \]
We want in the end:

*Well-initialzied 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:

\[
\begin{align*}
\text{vars } a & \subseteq \text{dom } s \implies \exists i. \text{aval } a \ s = \text{Some } i \\
\text{vars } b & \subseteq \text{dom } s \implies \exists \bv. \text{bval } b \ s = \text{Some } \bv \\
D \ A \ c \ A' & \implies A \subseteq A'
\end{align*}
\]
23 Definite Initialization Analysis

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

**Examples:** Is $x$ initially dead or live?

- $x := 0$ ☹
- $y := x; y := 0; x := 0$ ☺
- `WHILE b DO y := x; x := 1` ☺
At the end of a command, we may be interested in the value of *only some of the variables*, e.g. *only the global variables* at the end of a procedure.

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

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

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

\[ L \ SKIP \ X = X \]

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

\[ L \ (c_1 ;; c_2) \ X = L \ c_1 \ (L \ c_2 \ X) \]

\[ L \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ X = \]

\[ \text{vars } b \cup L \ c_1 \ X \cup L \ c_2 \ X \]

Example:

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

\[ \{"x"\} = \{"z"\} \]
WHILE b DO c

$L w X$ must satisfy

vars $b$ ⊆ $L w X$ (evaluation of $b$)

$X$ ⊆ $L w X$ (exit)

$L c (L w X)$ ⊆ $L w X$ (execution of $c$)
We define

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

\[ \implies \]

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

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

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

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

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

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

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

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

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

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

\texttt{gen SKIP} \quad = \quad \{\}

\texttt{gen (x ::= a)} \quad = \quad \text{vars } a

\texttt{gen (c_1;; c_2)} \quad = \quad \text{gen } c_1 \cup (\text{gen } c_2 - \text{kill } c_1)

\texttt{gen (IF b THEN c_1 ELSE c_2)} \quad =

\quad \text{vars } b \cup \text{gen } c_1 \cup \text{gen } c_2

\texttt{gen (WHILE b DO c)} \quad = \quad \text{vars } b \cup \text{gen } c
\[ L \ c \ X = gen \ c \cup (X - kill \ c) \]

Proof by induction on \( c \).

\[ \implies \]

\[ L \ c \ (L \ w \ X) \subseteq L \ w \ X \]
Digression: definite initialization via gen/kill

\[ A \ c \ X: \text{ the set of variables initialized after } c \]
if \( X \) was initialized before \( c \)

How to obtain \( A \ c \ X = X - \text{kill } c \cup \text{gen } c: \)

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

\( \text{kill } c = \{\} \)
Live Variable Analysis

Correctness of $L$

Dead Variable Elimination

True Liveness

Comparisons
\((.,.) \Rightarrow \) \quad \text{and } L \text{ should roughly be related like this:}

\textit{The value of the final state on } X \\
\textit{only depends on} \\
\textit{the value of the initial state on } L \cap X.

Put differently:

\textit{If two initial states agree on } L \cap X \\
\textit{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 \texttt{WHILE} cases we do not need the definition of $L_w$ but only the characteristic property

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

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

$L_w X$ should be the least set such that
\[ \text{vars } b \cup X \cup L_c (L_w X) \subseteq L_w X. \]

Follows easily from $L_c X = \text{gen } c \cup (X - \text{kill } c)$:
\[
\begin{align*}
\text{vars } b \cup X \cup L_c P & \subseteq P \\
L (\text{WHILE } b \text{ DO } c) X & \subseteq P
\end{align*}
\]
Live Variable Analysis

Correctness of $L$

Dead Variable Elimination

True Liveness

Comparisons
Bury all assignments to dead variables:

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

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

\textit{bury} \, c \, UNIV \sim c

where \textit{UNIV} is the set of all variables.

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

Follows from generalized statement:

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

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

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

Follows from generalized statement:

If  \((\text{bury } c \ X, \ s) \Rightarrow s'\) 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 \text{bury} for every kind of command, e.g.

\[(bc_1;; bc_2 = \text{bury } c \ X) = (\exists c_1 \ c_2.
\begin{align*}
&c = c_1;; c_2 \land \\
&bc_2 = \text{bury } c_2 \ X \land bc_1 = \text{bury } c_1 (L \ c_2 \ X))
\]
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 $x \leq y \implies f \ x \leq f \ y$ for all $x,y$, then $f$ is **monotone**.
Application to $L \ w$

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

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

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

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

Better:

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

But then

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

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

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

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 \( \text{distinct } [x, \ y, \ z] \)

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
\[ \lambda P. \mathit{vars} \ b \cup X \cup L \ c \ P \]
Existence of least fixpoints

**Theorem (Knaster-Tarski)** Let $f : \tau \text{ set} \Rightarrow \tau \text{ set}$. If $f$ is monotone ($X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$) then

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

is the least pre-fixpoint and least fixpoint of $f$. 
Proof of Knaster-Tarski

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

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

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

**Proof**
- \( f p \leq p \implies f p = p \)
- \( p \) is the least fixpoint
Definition of $L$

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

$L\ (\text{WHILE } b \ \text{DO } c)\ X = \text{lfp } f_w$
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:
\[ \text{lfp } f \subseteq \text{lfp } g \text{ if for all } X, f\ X \subseteq g\ X \]

**Corollary** $f_w$ is monotone.
**Computation of \( \text{lfp} \)**

**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 \( \text{lfp}(f) = f^k(\{\}) \).

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

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

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

Let $vars \ c$ be the variables in $c$.

**Lemma** $L \ c \ X \subseteq vars \ c \cup X$

**Proof** by induction on $c$

Let $V_w = vars \ b \cup vars \ c \cup X$

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

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

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

because $f_w \{\} \supseteq vars \ b \cup X$. 
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 \textit{lfp} in Isabelle

From the library theory \texttt{While\_Combinator}:

\texttt{while} :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
while \texttt{b \_ f \_ s} = (if \texttt{b \_ s} then \texttt{while b \_ f \_ (f \_ s)} else \texttt{s})

\textbf{Lemma} Let \texttt{f} :: \(τ \text{ set} \⇒ \tau \text{ set}\). If

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

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

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

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

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

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

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

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

$\forall P. \ f_w \ P \subseteq g_w \ P \implies f_w(g_w^i \ {}) = g_w(g_w^i \ {}) = g_w^i \ {}$

$\implies L \ w \ X = \text{lfp } f_w \subseteq g_w^i \ {} = Lb \ w \ X$

If $Lb \ w \ X = V_w$: $L \ w \ X \subseteq V_w$ (by Lemma)
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.
A Relational Denotational Semantics of IMP

Continuity
What is it?

A denotational semantics maps syntax to semantics:

\[ D :: syntax \Rightarrow meaning \]

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

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

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

Fake: \( Big\_step \ c = \{(s,t). \ (c,s) \Rightarrow t\} \)
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
Id :: (′a × ′a) set
Id = {p. ∃x. p = (x, x)}

op O :: (′a × ′b) set ⇒ (′b × ′c) set ⇒ (′a × ′c) set
r O s = {(x, z). ∃y. (x, y) ∈ r ∧ (y, z) ∈ s}
\[ D :: \text{com} \Rightarrow \text{com}\_\text{den} \]

**type_synonym** \[ \text{com}\_\text{den} = (\text{state} \times \text{state}) \text{ set} \]

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

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

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

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

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

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

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

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

Wanted:

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

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

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

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

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

\[ D \left( \text{WHILE } b \text{ DO } c \right) = \text{lfp} \left( W \left( b\text{val } b \right) \left( D \ c \right) \right) \]

By definition (where \( f = W \left( b\text{val } b \right) \left( D \ c \right) \)):

\[
D \ w = \text{lfp } f = f \left( \text{lfp } f \right) = W \left( b\text{val } v \right) \left( D \ c \right) \left( D \ w \right)
\]

\[
= \left\{ (s, t). \text{ if } b\text{val } b \ s \text{ then } (s, t) \in D \ c \ O \ D \ w \text{ else } s = t \right\}
\]
Why least?

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

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

Then

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

Every relation is a fixpoint!
Only the least relation \( \emptyset \) makes computational sense.
A denotational equivalence proof

Example

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

where \( w = WHILE \ b \ DO \ c. \)

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

\[ D \ w \]

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

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

Lemma \((c, s) \Rightarrow t \iff (s, t) \in D_c\)
Proof by rule induction

Lemma \((s, t) \in D_c \iff (s, t) \in \text{Big\_step } c\)
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**

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

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

**Definition (Continuous)**

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

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

**Lemma** \( \text{cont } f \implies \text{mono } f \)
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**

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

Semantics: $\{(s, t). \ 0 \leq s \ "x" \land t = s("x" := 0)\}$

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

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

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

**Lemma** \( single_{-}\text{valued } (D \ c) \)
Chapter 12

Hoare Logic
27 Partial Correctness

28 Verification Conditions

29 Total Correctness
27 Partial Correctness

28 Verification Conditions

29 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 + ... + 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*. 

---

*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*. 

---

*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*. 

---

*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:
- \{\} \ x := \ 5 \ \{ x = 5 \}
- \{\} \ x := \ x + 5 \ \{ x = 5 \}
- \{\} \ x := 2* (x + 5) \ \{ x > 20 \}

Intuitive explanation of backward-looking rule:

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

Afterwards we can replace all occurrences of \( a \) in \( Q \) by \( x \).
The assignment axiom allows us to compute the precondition from the postcondition.

There is a version to compute the postcondition from the precondition, but it is more complicated.
More rules of Hoare Logic

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

\[
\begin{align*}
\{P \land b\} \quad c_1 \quad \{Q\} \\
\{P \land \neg b\} \quad c_2 \quad \{Q\}
\end{align*}
\]

\[
\{P\} \quad IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}
\]

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

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

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

$$P' \rightarrow P \quad \{P\} \ c \ \{Q\} \quad Q \rightarrow Q'$$

$$\{P'\} \ c \ \{Q'\}$$

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.

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

\[
\{P \land b\} \quad \{P\} \quad P \land \neg b \rightarrow Q \quad \{P\} \quad \text{WHILE } b \text{ DO } c \quad \{Q\}
\]
\{ x = i \}

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

\{ y = \textit{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: *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

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

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

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

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

\[ = (\lambda s. \ s \ x + 5 = 5) \]
\[ \vdash \{ P \} \quad c_1 \quad \{ Q \} \quad \vdash \{ Q \} \quad c_2 \quad \{ R \} \]

\[ \vdash \{ P \} \quad c_1 ; ; \quad c_2 \quad \{ R \} \]

\[ \vdash \{ \lambda s. \ P \ s \ \land \ bval \ b \ s \} \quad c_1 \quad \{ Q \} \]
\[ \vdash \{ \lambda s. \ P \ s \ \land \ \neg \ bval \ b \ s \} \quad c_2 \quad \{ Q \} \]

\[ \vdash \{ \ P \} \quad IF \ b \ THEN \ c_1 \ ELSE \ c_2 \quad \{ Q \} \]

\[ \vdash \{ \lambda s. \ P \ s \ \land \ bval \ b \ s \} \quad c \quad \{ P \} \]

\[ \vdash \{ P \} \quad WHILE \ b \ DO \ c \quad \{ \lambda s. \ P \ s \ \land \ \neg \ bval \ b \ s \} \]
\forall s. \quad P' \ s \ \rightarrow \ P \ s
\vdash \{ P \} \ c \ \{ Q \}
\forall s. \quad Q \ s \ \rightarrow \ Q' \ s
\vdash \{ P' \} \ c \ \{ Q' \}
Hoare_Examples.thy
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: $\vdash \implies \top$
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 $wp$

$\text{wp } \text{SKIP } Q = Q$

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

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

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

$\neg \ bval \ b \ s \rightarrow \text{wp } (\text{WHILE } b \ \text{DO } c) \ Q \ s = Q \ s$

$bval \ b \ s \rightarrow$

$\text{wp } (\text{WHILE } b \ \text{DO } c) \ Q \ s =$

$\text{wp } (c;; \ \text{WHILE } b \ \text{DO } c) \ Q \ s$
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 \( wp \).
$\vdash \{ P \} \ c \ \{ Q \} \iff \models \{ P \} \ c \ \{ Q \}$

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

**WARNING**
Partial Correctness

Verification Conditions

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 see Chapter 12).
Terminology:

**VCG** = Verification Condition Generator

All successful verification technology for imperative programs relies on
- VCGs (of one kind or another)
- and powerful (semi-)automatic theorem provers.
The (approx.) plan of attack

1. Introduce annotated commands with loop invariants

2. Define functions for *computing*
   - weakest preconditions: $\text{pre} :: \text{com} \Rightarrow \text{assn} \Rightarrow \text{assn}$
   - verification conditions: $\text{vc} :: \text{com} \Rightarrow \text{assn} \Rightarrow \text{bool}$

3. Soundness: $\text{vc} \ c \ Q \iff \vdash \{ ? \} \ c \ \{ Q \}$

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` = `Skip`

<table>
<thead>
<tr>
<th><code>Assign vname aexp</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Seq acom acom</code></td>
</tr>
<tr>
<td><code>If bexp acom acom</code></td>
</tr>
<tr>
<td><code>While assn bexp acom</code></td>
</tr>
</tbody>
</table>

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

```
{I} WHILE b DO c
```
**Weakest precondition**

\[ \text{pre} :: \text{acom} \Rightarrow \text{assn} \Rightarrow \text{assn} \]

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

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

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

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

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

In the presence of loops, $\text{pre } C$ 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

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

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

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

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

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

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

\[ vc \ C \ Q \iff \vdash \{ pre \ C \ Q \} \ \text{strip} \ C \ \{ Q \} \]

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

Corollary:

\[
\begin{array}{c}
[vc \ C \ Q; \ \forall s. \ P \ s \implies pre \ C \ Q \ s] \\
\implies \vdash \{P\} \ \text{strip} \ C \ \{ Q \}
\end{array}
\]

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

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

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

Why could premise not be provable although conclusion is?

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

Therefore we prove completeness: suitable annotations exist such that premise is provable.
Completeness of \( vc \) \& \( pre \) w.r.t. \( \vdash \)

\[ \vdash \{ P \} \ c \ \{ Q \} \ \Rightarrow \exists \ C. \ \text{strip} \ C = c \ \land \ vc \ C \ Q \ \land \ (\forall \ s. \ P \ s \ \rightarrow \ \text{pre} \ C \ Q \ s) \]

Proof by rule induction. Needs two monotonicity lemmas:

\[ [\forall \ s. \ P \ s \ \rightarrow \ P' \ s; \ \text{pre} \ C \ P \ s] \ \Rightarrow \ \text{pre} \ C \ P' \ s \]

\[ [\forall \ s. \ P \ s \ \rightarrow \ P' \ s; \ vc \ C \ P] \ \Rightarrow \ vc \ C \ P' \]
Partial Correctness

Verification Conditions

Total Correctness
- Partial Correctness: 
  \textit{if} command terminates, postcondition holds
- Total Correctness: 
  command terminates \textit{and} postcondition holds

\begin{align*}
\text{Total Correctness} & = \text{Partial Correctness} + \text{Termination}
\end{align*}

Formally:

\begin{align*}
(\models_{t} \{P\} c \{Q\}) & = \\
(\forall s. \ P\ s \rightarrow (\exists t. (c, s) \Rightarrow t \land Q \ t))
\end{align*}

Assumes that semantics is deterministic!

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

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

Some measure function \textit{state} ⇒ \textit{nat} must decrease with every loop iteration

\[
\bigwedge n. \vdash_t \{ \lambda s. P s \land bval b s \land n = f s \} \ C \{ \lambda s. P s \land f s < n \}
\]

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

\[
\forall n. \frac{\{\lambda s. P \ s \land bval \ b \ s \land T \ s \ n\} \ c \ \{\lambda s. P \ s \land (\exists n'<n. \ T \ s \ n')\}}{\frac{\{\lambda s. P \ s \land (\exists n. \ T \ s \ n)\} \ \text{WHILE} \ b \ \text{DO} \ c \ \{\lambda s. P \ s \land \neg bval \ b \ s\}}}
\]
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

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

where

$$\wp_t \ c \ Q = (\lambda s. \ \exists \ t. \ (c, \ s) \Rightarrow t \land Q \ t).$$
Proof of $\vdash_t \{ \text{wp}_t \ c \ Q \} \ c \ \{ Q \}$ is by induction on $c$.

In the $\text{WHILE } b \ DO \ c$ case, use the $\text{WHILE}$ rule with

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

$T \ s \ n$ means that $\text{WHILE } b \ DO \ c$ 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
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination
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:
  \[ a[i] := 1; a[j] := 2; x := a[i] \]
  \[ a[i] := 1; a[j] := 2; x := 1 \]
  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

- ...
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.
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

\[
\begin{align*}
\{ \text{Inv} \} \\
\text{WHILE } b \text{ DO } \{ P \} \ c \\
\{ Q \}
\end{align*}
\]

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

Collects all possible states for each program point:

\begin{verbatim}
x := 0 \{ <x := 0> \} ;
{ <x := 0>, <x := 2>, <x := 4> }
WHILE x < 3
DO { <x := 0>, <x := 2> }
x := x+2 \{ <x := 2>, <x := 4> \}
{ <x := 4> }
\end{verbatim}
Infinite sets of states:

\{\ldots, <x := -1>, <x := 0>, <x := 1>, \ldots \} \\
WHILE x < 3 \\
DO \{\ldots, <x := 1>, <x := 2> \} \\
\quad x := x+2 \{\ldots, <x := 3>, <x := 4> \} \\
\{ <x := 3>, <x := 4>, \ldots \}
Multiple variables:

\[
\begin{align*}
x & := 0; \ y := 0 \quad \{ <x:=0, \ y:=0> \} ; \\
\} \quad \{ <x:=0, \ y:=0>, \ <x:=2, \ y:=1>, \ <x:=4, \ y:=2> \} \\
\text{WHILE x < 3} \\
\text{DO } \{ <x:=0, \ y:=0>, \ <x:=2, \ y:=1> \} \\
\quad x := x+2; \ y := y+1 \\
\quad \{ <x:=2, \ y:=1>, \ <x:=4, \ y:=2> \} \\
\} \quad \{ <x:=4, \ y:=2> \}
\end{align*}
\]
A first approximation

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

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

\[
\text{WHILE } x < 3 \text{ DO} \begin{cases} <x := \{0,2\}> \end{cases} \\
x := x+2 \begin{cases} <x := \{2,4\}> \end{cases} \\
\{ <x := \{4\}> \}
\]
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

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

Approximate sets of concrete values by *abstract values*

Example: approximate sets of numbers by intervals

Execute/interpret program with abstract values
Example

Consistently annotated program:

\[
\begin{align*}
x & := 0 \{ \langle x := [0,0] \rangle \} ; \\
\{ \langle x := [0,4] \rangle \} \\
\text{WHILE } x < 3 \\
\text{DO } \{ \langle x := [0,2] \rangle \} \\
\quad x & := x + 2 \{ \langle x := [2,4] \rangle \} \\
\{ \langle x := [3,4] \rangle \}
\end{align*}
\]

The annotations are computed by

- starting from an un-annotated program and
- iterating abstract execution
- until the annotations stabilize.
x := 0

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

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination
Concrete syntax

'a acom ::= SKIP { 'a } | string ::= aexp { 'a }
| 'a acom ;; 'a acom
| IF bexp THEN { 'a } 'a acom ELSE { 'a } 'a acom
{ 'a }
| { 'a }
WHILE bexp DO { 'a } 'a acom
{ 'a }

'a: type of annotations

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

datatype

\[ \texttt{\}'a\ acom = SKIP \ 'a } \]

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

Strips all annotations from an annotated command

\begin{align*}
\text{strip} :: 'a \, a\text{com} & \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 THEN} \ \{P_1\} \ C_1 \ \text{ELSE} \ \{P_2\} \ C_2 \ \{P\}) & = \text{IF b THEN} \ \text{strip} \ C_1 \ \text{ELSE} \ \text{strip} \ C_2 \\
\text{strip} \ (\{I\} \ \text{WHILE b DO} \ \{P\} \ C \ \{Q\}) & = \text{WHILE b DO} \ \text{strip} \ C
\end{align*}

We call $C$ and $C'$ \textit{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)

\textit{annos} :: 'a acom ⇒ 'a list

\begin{align*}
\text{annos } (\text{SKIP} \{ P \}) & = [P] \\
\text{annos } (x ::= e \{ P \}) & = [P] \\
\text{annos } (C_1 ;; C_2) & = \text{annos } C_1 @ \text{annos } C_2 \\
\text{annos } (\text{IF } b \text{ THEN } \{ P_1 \} C_1 \text{ ELSE } \{ P_2 \} C_2 \{ Q \}) & = P_1 \# \text{annos } C_1 @ P_2 \# \text{annos } C_2 @ [Q] \\
\text{annos } (\{ I \} \text{ WHILE } b \text{ DO } \{ P \} C \{ Q \}) & = I \# P \# \text{annos } C @ [Q]
\end{align*}
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 \text{ acom} & \Rightarrow 'a \\
\text{post } C & = \text{ last } (\text{annos } C)
\end{align*}

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

\begin{align*}
\text{map\_acom} &:: (\mathbf{a} \Rightarrow \mathbf{b}) \Rightarrow \mathbf{a} \text{ acom} \Rightarrow \mathbf{b} \text{ acom} \\
\text{map\_acom } f \ C &\quad \text{applies } f \text{ to all annotations in } C
\end{align*}
30 Introduction

31 Annotated Commands

32 Collecting Semantics

33 Abstract Interpretation: Orderings

34 A Generic Abstract Interpreter

35 Executable Abstract State

36 Termination
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
\[ \text{step } S \left( \text{SKIP } \{ - \} \right) = \text{SKIP } \{ S \} \]

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

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

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

View command as a control flow graph

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

Stabilization means fixpoint:

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

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

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

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

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

\text{Fixpoint = solution of equation system}

\text{Iteration is just one way of solving equations}
Why *least* fixpoint?

\[
\{ \{ I \} \}
\]

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

\{ \{ \} \}

Is fixpoint of \textit{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 \( \leq : 'a \Rightarrow 'a \Rightarrow \text{bool} \)
- that is reflexive \( (x \leq x) \),
- transitive \( ([x \leq y; y \leq z] \Rightarrow x \leq z) \) and
- antisymmetric \( ([x \leq y; y \leq x] \Rightarrow x = y) \)
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 (infimum) of \( S \) is often denoted by \( \bigcap S \).

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

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

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

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

**Definition** A function \( f \) on a partial order \( \leq \) is **monotone** if \( x \leq y \implies f x \leq f y \).

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

\[
\bigsqcap \{ p. f p \leq p \}.
\]

**Proof** just like the version for sets.
An ordering on \( 'a \) can be lifted to \( 'a \ acom \) by comparing the annotations of \( \text{strip} \)-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:

\[
x ::= N 0 \{\{a\}\} \leq x ::= N 0 \{\{a, b\}\} \quad \leftrightarrow \quad True
\]
\[
x ::= N 0 \{\{a\}\} \leq x ::= N 0 \{\{}\} \quad \leftrightarrow \quad False
\]
\[
x ::= N 0 \{S\} \leq x ::= N 1 \{S\} \quad \leftrightarrow \quad False
\]
The collecting semantics needs to order $state\ set\ acom$. Annotations are (state) sets ordered by $\subseteq$, which form a complete lattice.

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

Almost...
A complication

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

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

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

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 \( L :: 'a \text{ set} \) 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 \( L :: 'a \text{ set} \) 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

$$\prod \{ SKIP \{ A\}, \text{SKIP} \{ B\}, \ldots \}$$

$$= \text{SKIP} \{ \prod \{ A,B, \ldots \} \}$$

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

\begin{align*}
\text{annotate} &:: (\text{nat} \Rightarrow \text{'a}) \Rightarrow \text{com} \Rightarrow \text{'a acom} \\
\text{Set annotation number } p \text{ (as counted by } \text{anno} \text{) to } f \; p. \\
\text{Definition is technical. The characteristic lemma:} \\
\text{anno } (\text{annotate } f \; c) \; p &= f \; p
\end{align*}
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

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

Proof straightforward (pointwise).
The Collecting Semantics

The underlying complete lattice is now *state set*.

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

**Lemma** $\text{step } S \in L \rightarrow L$ and is monotone.

Therefore Knaster-Tarski is applicable and we define

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

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

Follows directly from

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

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

uses

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

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.
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination
Approximating the Collecting semantics

A conceptual step:

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

A domain-specific step:

\[val \text{ set} \sim \text{ 'av}\]

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

Abstract values:

**datatype** *parity* = *Even* | *Odd* | *Either*

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.
Top element

A partial order $\mathcal{P}$ has a top element $\top :: \mathcal{P}$ if

$$a \leq \top$$
A type \( 'a \) is a semilattice if
- it is a partial order and
- there is a least upper bound operation

\[
\bigcirc :: 'a \Rightarrow 'a \Rightarrow 'a
\]

\[
x \leq x \bigcirc y \quad y \leq x \bigcirc y
\]

\[
\left[ x \leq z; y \leq z \right] \implies x \bigcirc y \leq z
\]

Application: abstract \( \bigcirc \), join two computation paths
We often call \( \bigcirc \) the \textit{join} operation.
\textbf{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 \( u_2 \leq u_1 \).
Example: parity

Fact Type *parity* is a semilattice with top element.
Isabelle’s type classes

A type class is defined by
- a set of required functions (the interface)
- and a set of axioms about those functions

Examples

class order: partial orders
class semilattice_sup: semilattices
class semilattice_sup_top: semilattices with top element

A type belongs to some class if
- the interface functions are defined on that type
- and satisfy the axioms of the class (proof needed!)

Notation: $\tau :: C$ means type $\tau$ belongs to class $C$
Example: parity :: semilattice_sup
HOL/Orderings.thy
Abs_Int1_parity.thy

Orderings and instances
From abstract values to abstract states

Need to abstract collecting semantics:

\[ \text{state set} \]

- First attempt:
  \[ 'av \text{ st } = \ u\text{name} \Rightarrow 'av \]
  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 $'a :: \text{semilattice\_sup\_top}$
then $'b \Rightarrow 'a :: \text{semilattice\_sup\_top}$

Proof

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

definition

$\gamma\_fun :: ('a \Rightarrow 'c\ \text{set}) \Rightarrow ('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'c)\text{set}$

where $\gamma\_fun\ \gamma\ F = \{f. \forall x. f\ x \in \gamma\ (F\ x)\}$

Lemma

If $\gamma$ is monotone then $\gamma\_fun\ \gamma$ is monotone.
Lemma  If 'a :: semilattice_sup_top
       then 'a option :: semilattice_sup_top

Proof
(\text{Some } x \leq \text{Some } y) = (x \leq y)
(\text{None } \leq _) = \text{True}
(\text{Some } _ \leq \text{None}) = \text{False}

\text{Some } x \sqcup \text{Some } y = \text{Some } (x \sqcup y)
\text{None } \sqcup y = y
x \sqcup \text{None} = x
\top = \text{Some } \top

Corollary  If 'a :: semilattice_sup_top
             then 'a st option :: semilattice_sup_top
fun \( \gamma_{\text{option}} :: (\alpha \Rightarrow \text{c set}) \Rightarrow \alpha \text{ option} \Rightarrow \text{c set} \)

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 \( 'a \ acom :: \text{order} \).

Partial order is enough, semilattice not needed.

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

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

**Lemma**

If \( \gamma \) is monotone then \( \text{map}_{acom} \gamma \) is monotone.
• Stepwise development of a generic abstract interpreter as a parameterized module
• Parameters/Input: abstract type of values together with abstractions of the operations on concrete type $\text{val} = \text{int}$.
• Result/Output: abstract interpreter that approximates the collecting semantics by computing on abstract values.
• Realization in Isabelle as a locale
Abstract values: type \( \texttt{av} :: \text{semilattice\textunderscore sup\textunderscore top} \)

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

Assumptions: \( a \leq b \implies \gamma a \subseteq \gamma b \)
\( \gamma \top = \text{UNIV} \)
Parameters (II)

Abstract arithmetic:  

\[ \text{num'} :: \text{val} \Rightarrow 'av \]
\[ \text{plus'} :: 'av \Rightarrow 'av \Rightarrow 'av \]

Intention:  

\( \text{num'} \) abstracts the meaning of \( N \)
\( \text{plus'} \) abstracts the meaning of \( \text{Plus} \)

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

Assumptions:

\[ i \in \gamma (\text{num'}i) \]
\[ [i_1 \in \gamma a_1; i_2 \in \gamma a_2] \implies i_1 + i_2 \in \gamma (\text{plus'}a_1 a_2) \]

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

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

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

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

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

fun $aval' :: aexp \Rightarrow 'av \ st \Rightarrow 'av$

$aval' (N \ n) \ S = \ num' \ n$

$aval' (V \ x) \ S = S \ x$

$aval' (Plus \ a_1 \ a_2) \ S = \ plus' (aval' \ a_1 \ S) (aval' \ a_2 \ S)$

Correctness of $aval'$ wrt $aval$:

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

**Proof** by induction on $a$

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

\[ \leq / \sqcup \text{ and } \gamma_{\text{parity}}: \text{ see earlier} \]

\[
num_{\text{parity}} i = (\text{if } i \mod 2 = 0 \text{ then } \text{Even} \text{ else } \text{Odd})
\]

\[
\text{plus}_{\text{parity}} \text{ Even Even} = \text{Even} \\
\text{plus}_{\text{parity}} \text{ Odd Odd} = \text{Even} \\
\text{plus}_{\text{parity}} \text{ Even Odd} = \text{Odd} \\
\text{plus}_{\text{parity}} \text{ Odd Even} = \text{Odd} \\
\text{plus}_{\text{parity}} \text{ Either } y = \text{Either} \\
\text{plus}_{\text{parity}} \text{ x Either} = \text{Either}
\]
Example instantiation with \textit{parity}

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

Must prove parameter assumptions

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

Example The value of

\begin{align*}
\text{aval}_{\text{parity}} \left( \text{Plus} \left( V "x" \right) \left( V "x" \right) \right) \\
\left( \lambda_. \text{Either} \left( "x" := \text{Odd} \right) \right)
\end{align*}

is \textit{Even}. 
Abs_Int1_parity.thy

Locale interpretation
Abstract interpretation of $bexp$

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

Abstracting the collecting semantics

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

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

to

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

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

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

\[
\text{Step} :: \text{'a} \Rightarrow \text{'a acom} \Rightarrow \text{'a acom}
\]
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
\[
\text{Step } a (\text{SKIP } \{\_\}) = \text{SKIP } \{a\}
\]

\[
\text{Step } a (x ::= e \{\_\}) = x ::= e \{\text{asem } x e a\}
\]

\[
\text{Step } a (C_1 ;; C_2) = \text{Step } a C_1 ;; \text{Step } (\text{post } C_1) C_2
\]

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

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

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

\[
\text{Step } \text{asem } \text{bsem} \ldots
\]

\[
\begin{align*}
\text{step} &=
\text{Step } (\lambda x \ e \ S. \ \{s(x := \text{aval } e \ s) \mid s. \ s \in S\}) \\
& \quad (\lambda b \ S. \ \{s \in S. \ \text{bval } b \ s\})
\end{align*}
\]

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

where

\[
\text{asem } x \ e \ S =
\begin{cases} 
\text{case } S \text{ of } \text{None } \Rightarrow \text{None} \\
\text{| Some } S \Rightarrow \text{Some } (S(x := \text{aval}' e \ S)) 
\end{cases}
\]
Example: iterating \textit{step\_parity}

$$(\text{step\_parity } S)^k \ c$$

where

$$c = x ::= N \ 3 \ \{\text{None}\} \ ; \ \{\text{None}\}$$

$$\text{WHILE } b \ \text{DO } \{\text{None}\}$$

$$x ::= \text{Plus} (V \ x) (N \ 5) \ \{\text{None}\} \ \{\text{None}\}$$

$$S = \text{Some} (\lambda_. \text{Either})$$

$$S_p = \text{Some} (((\lambda_. \text{Either})(x := p)))$$
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** $Step \ f \ g \ (\gamma_o \ S) \ (\gamma_c \ C) \leq \gamma_c \ (Step \ f' \ g' \ S \ C)$

if for all $x, e, b$: $f \ x \ e \ (\gamma_o \ S) \subseteq \gamma_o \ (f' \ x \ e \ S)$

$g \ b \ (\gamma_o \ S) \subseteq \gamma_o \ (g' \ b \ S)$

**Proof** by an easy induction on $C$
The abstract interpreter

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

From the HOL library:

\[ \text{while_option} :: \]
\[ (\mathbf{a} \Rightarrow \mathbf{bool}) \Rightarrow (\mathbf{a} \Rightarrow \mathbf{a}) \Rightarrow \mathbf{a} \Rightarrow \mathbf{a} \text{ option} \]

such that

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

and \[ \text{while_option } b \ f \ x = \text{None} \]
if the recursion does not terminate.
Pre-fixpoint:

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

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

Start iteration with least annotated command:

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

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

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

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

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

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination
Solution

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

An association list representation:

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

From `'a st_rep` back to `vname ⇒ 'a`:

```plaintext
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”:

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

Not a partial order because not antisymmetric!

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

Define \( \text{eq\_st \ ps_1 \ ps_2 = (fun\_rep \ ps_1 = fun\_rep \ ps_2) } \)

Overwrite 'a st = vname \( \Rightarrow \) 'a by 

**quotient\_type** 'a st = ('a::top) st\_rep / eq\_st

Elements of 'a st:

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

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:
\[
\text{fun } (St \ ps) = \text{fun_rep } ps
\]

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

\[(St\ ps_1 \leq St\ ps_2) = \text{less}_\text{eq}_\text{st}_\text{rep}\ ps_1\ ps_2\]

\[St\ ps_1 \sqcup\ St\ ps_2 = St(\text{map}_2\text{st}_\text{rep}\ (op\ \sqcap)\ ps_1\ ps_2)\]

fun \text{map}_2\text{st}_\text{rep} ::
('a::top ⇒ 'a ⇒ 'a) ⇒ 'a\text{st}_\text{rep} ⇒ 'a\text{st}_\text{rep} ⇒ 'a\text{st}_\text{rep}

\text{map}_2\text{st}_\text{rep}\ f\ ((x,\ y)\ #\ ps_1)\ ps_2\ =
let\ y_2 = \text{fun}_\text{rep}\ ps_2\ x
in\ (x,\ f\ y\ y_2)\ #\ \text{map}_2\text{st}_\text{rep}\ f\ ps_1\ ps_2

\text{map}_2\text{st}_\text{rep}\ f\ []\ ps_2 = \text{map}\ (\lambda(x,\ y).\ (x,\ f\ ⊤\ y))\ ps_2
Modified abstract interpreter

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

\[
\begin{align*}
S \ x & \leadsto \ \text{fun } S \ x \\
S(x := a) & \leadsto \ \text{update } S \ x \ a
\end{align*}
\]

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

Examples
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Executable Abstract State

Termination
Beyond partial correctness

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

The solution: Monotonicity

$\Rightarrow$

**Precision** $AI$ computes *least* pre-fixpoints

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

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

\[
[a_1 \leq b_1; \ a_2 \leq b_2] \implies plus' \ a_1 \ a_2 \leq plus' \ b_1 \ b_2
\]

**Theorem** In the monotone framework, \(aval'\) is also monotone

\[
S_1 \leq S_2 \implies aval' \ e \ S_1 \leq aval' \ e \ S_2
\]

and therefore \(step'\) is also monotone:

\[
[S_1 \leq S_2; \ C_1 \leq C_2] \implies step' \ S_1 \ C_1 \leq step' \ S_2 \ C_2
\]
Precision: smaller is better

If $f$ is monotone and $\bot$ is a least element, then $pfp f \bot$ is a least pre-fixpoint of $f$. 
Lemma Let $\leq$ be a partial order on a set $L$ with least element $\bot \in L$: $x \in L \implies \bot \sqsubseteq x$.
Let $f \in L \to L$ be a monotone function.
If \texttt{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 \textit{bot} \ c \ is \ a \ least \ element \ and \ \textit{step}' \ is \ monotone, \ \textit{AI} \ returns \ least \ pre-fixpoints
Because $\text{step}'$ is monotone, starting from $\text{bot } c$ generates an ascending $<$ chain of annotated commands. We exhibit a measure function $m_c$ that decreases with every loop iteration:

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

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

Parameters:  

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

$$h :: nat$$

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:

\[
\begin{align*}
\text{finite } X & \implies m_s S X \leq h \times \text{card } X \\
\text{finite } X & \implies m_o \text{ opt } X \leq h \times \text{card } X + 1 \\
m_c C & \leq \text{length } (\text{annos } C) \times (h \times \text{card } (\text{vars } C) + 1)
\end{align*}
\]

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:

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

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 \( X \).
\[
\text{top}_\text{on}
\]

\[
top\_on\text{s} :: \text{'av st} \Rightarrow \text{vname set} \Rightarrow \text{bool}
\]
\[
top\_on\text{s} S X = (\forall x \in X. \ S x = \top)
\]

\[
top\_on\text{o} :: \text{'av st option} \Rightarrow \text{vname set} \Rightarrow \text{bool}
\]
\[
top\_on\text{o} (\text{Some } S) X = top\_on\text{s} S X
\]
\[
top\_on\text{o} \text{ None } X = \text{True}
\]

\[
top\_on\text{c} :: \text{'av st option acom} \Rightarrow \text{bool}
\]
\[
top\_on\text{c} C X = (\forall a \in \text{set (annos } C). \ top\_on\text{o} a X)
\]
Now we can formulate and prove anti-monotonicity:

\[
\begin{align*}
&\left[\text{finite } X;\ S_1 = S_2 \text{ on } -X;\ S_1 < S_2\right] \\
&\implies m_s S_2 X < m_s S_1 X
\end{align*}
\]

\[
\begin{align*}
&\left[\text{finite } X;\ \text{top}_o o_1 (-X);\ \text{top}_o o_2 (-X);\ o_1 < o_2\right] \\
&\implies m_o o_2 X < m_o o_1 X
\end{align*}
\]

\[
\begin{align*}
&\left[\text{top}_c C_1 (-\text{vars } C_1);\ \text{top}_c C_2 (-\text{vars } C_2);\ C_1 < C_2\right] \\
&\implies m_c C_2 < m_c C_1
\end{align*}
\]
Now we can prove termination

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

because \textit{step}' leaves \textit{top} \_\textit{on} \_\textit{s} invariant:

$$top\_on_c \ C (- vars \ C) \Rightarrow$$
$$top\_on_c \ (step' \top \ C) (- vars \ C)$$
Warning: \textit{step'} is very inefficient. It is applied to every subcommand in every step. Thus the actual complexity of $AI$ 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: efficiency complicates proofs.
Abs_Int1_parity.thy
Abs_Int1_const.thy

Termination