Concrete Semantics

—

A Proof Assistant Based Approach

Tobias Nipkow

Fakultät für Informatik
TU München

Wintersemester 2011
Introduction
1 Introduction
1 Introduction

Background

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!
- I assume you have the necessary intuition.
- This course is about “beyond intuition”.

Intuition is not sufficient!

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

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

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

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

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

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

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

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

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

Robin Milner (1934–2010)
Turing Award 1991.

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

SML semantics hardly used:

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

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

Machine-checked language semantics and proofs

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

The tool:

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

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

Government health warnings:

- Time consuming
- Potentially addictive
- Undermines your naive trust in informal proofs
Terminology

This lecture course:

Formal = machine-checked
Verification = formal correctness proof

Traditionally:

Formal = mathematical
Two landmark verifications

C compiler
Competitive with gcc -O1

Xavier Leroy
INRIA Paris
using Coq

Operating system
microkernel (L4)

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

Programming language researchers are increasingly using PAs
Why verification pays off

Short term: *The software works!*

Long term:

Tracking effects of changes by rerunning proofs

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

Long term much more important than short term:

*Software Never Dies*
1 Introduction
   Background
   This Course
What this course is *not* about

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

- Techniques for the description and analysis of
  - PLs
  - PL tools
  - Programs

- Description techniques: *operational semantics*
- Proof techniques: *inductions*

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

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

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

All exercises require the use of Isabelle/HOL
Why I am so passionate about the PA part

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

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

The use of a PA is leading edge

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

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

It is a new approach to informatics
Part I

Programming and Proving in HOL
2 Overview of Isabelle/HOL
3 Type and function definitions
4 Induction and Simplification
5 Case Study: IMP Expressions
6 Logic and Proof beyond “=”
7 Isar: A Language for Structured Proofs
Implication associates to the right:

\[ A \implies B \implies C \] means \[ A \implies (B \implies C) \]

Similarly for other arrows: \( \triangleright, \quad \triangleright\rightarrow \)

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

3 Type and function definitions

4 Induction and Simplification

5 Case Study: IMP Expressions

6 Logic and Proof beyond “=”

7 Isar: A Language for Structured Proofs
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 $\text{term} = \text{term}$,
  
e.g. $1 + 2 = 4$
- Later: $\land$, $\lor$, $\rightarrow$, $\forall$, …
Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types bool, nat and list

Summary
Types

Basic syntax:

\[ \tau ::= (\tau) \]
\[ | \text{bool} | \text{nat} | \text{int} | \ldots \]
\[ | 'a | 'b | \ldots \]
\[ | \tau \Rightarrow \tau \]
\[ | \tau \times \tau \]
\[ | \tau \text{ list} \]
\[ | \tau \text{ set} \]
\[ | \ldots \]

Convention:

\[ \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3) \]
Terms can be formed as follows:

- **Function application:**
  \[ f \, t \]
  is the call of function \( f \) with argument \( t \).
  If \( f \) has more arguments: \( f \, t_1 \, t_2 \, \ldots \)
  Examples: \( \sin \pi, \plus \, x \, y \)

- **Function abstraction:**
  \[ \lambda x. \, t \]
  is the function with parameter \( x \) and result \( t \), i.e. \( \text{“} x \mapsto t \text{”} \).
  Example: \( \lambda x. \, \plus \, x \, x \)
Terms

Basic syntax:

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

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 \( \tau \)”.

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

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

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

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

Thou shalt Curry your functions

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

Advantage:

Currying allows *partial application*

$f a_1$ where $a_1 :: \tau_1$
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

\[
\| f \, x + y \equiv (f \, x) + y \not\equiv f (x + y) \| 
\]

Enclose if and case in parentheses:

\[
\| (if \, \_ \, then \, \_ \, else \, \_ ) \| 
\]
Isabelle text = Theory = Module

Syntax: theory MyTh
imports ImpTh₁ ... ImpThₙ
begin
(definitions, theorems, proofs, ...)*
end

MyTh: name of theory. Must live in file MyTh.thy
ImpThᵢ: name of imported theories. Import transitive.

Usually: imports Main
Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types bool, nat and list

Summary
Proof General

An Isabelle Interface

by David Aspinall
Proof General

Customized version of (x)emacs:

- all of emacs
- Isabelle aware (when editing .thy files)
- mathematical symbols ("x-symbols")
  (eg $\iff$ instead of $\Rightarrow$, $\forall$ instead of ALL)
Similar to ProofGeneral but

- based on jedit
- \( \Rightarrow \) easier to install
- \( \Rightarrow \) may be more familiar
- Has advantages and a few disadvantages
Concrete syntax

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

Except for single identifiers

" normally not shown on slides
Overview_Demo.thy
Overview of Isabelle/HOL

Types and terms

Interfaces

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

Summary
**Type** `bool`

**datatype**  
`bool = True | False`

Predefined functions:  
`∧, ∨, →, ... :: bool ⇒ bool ⇒ bool`

A logical formula is a term of type `bool`

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

\textbf{datatype} \hspace{1em} \textit{nat} = 0 \mid \text{Suc} \hspace{0.5em} \textit{nat}

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

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

! Numbers and arithmetic operations are overloaded:
0, 1, 2, \ldots :: 'a, \hspace{1em} + :: 'a \Rightarrow 'a \Rightarrow 'a

You need type annotations: 1 :: \textit{nat}, x + (y :: \textit{nat}) unless the context is unambiguous: \text{Suc} \hspace{0.5em} z
Nat_Demo.thy
An informal proof

**Lemma** \( add\ m\ 0 = m \)

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

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

- **Case** \( Suc\ m \) (the induction step):
  We assume \( add\ m\ 0 = m \), the induction hypothesis (IH), and 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)

Syntactic sugar:

- `[] = Nil: empty list
- `x # xs = Cons x xs: list with first element `x ( "head") and rest `xs ( "tail")
- `[x_1, \ldots, x_n] = x_1 # \ldots x_n # []
Structural Induction for lists

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

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

\[
\begin{array}{c}
P([], \bigwedge x \; xs. \; P(xs) \implies P(x\#xs))
\end{array}
\]
List_Demo.thy
Lemma \( \text{app} (\text{app} \ \text{xs} \ \text{ys}) \ \text{zs} = \text{app} \ \text{xs} (\text{app} \ \text{ys} \ \text{zs}) \)

Proof by induction on \( \text{xs} \).

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

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

The proof is as follows:

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

Included in Main.

Don’t reinvent, reuse!

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

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

\textbf{fun} \quad \texttt{map} :: \ (\mapsto a)\Rightarrow\ a\ list\Rightarrow\ b\ list \quad \textbf{where}

\begin{align*}
\textbf{fun} \quad \texttt{map} :: \ (\mapsto a)\Rightarrow\ a\ list\Rightarrow\ b\ list & \quad \textbf{where} \\
\texttt{map} \ f\ [] & = [] \\
\texttt{map} \ f\ (x\#xs) & = f\ x\ #\ \texttt{map} \ f\ xs
\end{align*}

Note: $map$ takes \textit{function} as argument.
Overview of Isabelle/HOL

Types and terms
Interfaces
By example: types bool, nat and list

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

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

- \textit{induction} performs structural induction on some variable (if the type of the variable is a datatype).

- \textit{auto} solves as many subgoals as it can, mainly by simplification (symbolic evaluation):
  
  \textit{“=”} is used only from left to right!
Proofs

General schema:

\textbf{lemma} \textit{name}: "..."
\textbf{apply} (...) \\
\textbf{apply} (...) \\
... \\
\textbf{done}

If the lemma is suitable as a simplification rule:

\textbf{lemma} \textit{name}[simp]: "..."
Top down proofs

Command

```
sorry
```

“completes” any proof.

Allows top down development:

Assume lemma first, prove it later.
The proof state

1. $\wedge x_1 \ldots x_p \cdot A \implies B$

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

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

abbreviates

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

; \quad \approx \quad "and"
2 Overview of Isabelle/HOL

3 Type and function definitions

4 Induction and Simplification

5 Case Study: IMP Expressions

6 Logic and Proof beyond “=”

7 Isar: A Language for Structured Proofs
Type and function definitions

Type definitions

Function definitions
Type synonyms

**type_synonym** *name* = *τ*

Introduces a *synonym* *name* for type *τ*

Examples:

**type_synonym** *string* = *char list*

**type_synonym** (*'a',*'b*)*foo* = *'a list* × *'b list*

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

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

- **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*:

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

Wildcards:  

\[
(case \, m \, of \, 0 \Rightarrow \text{Suc} \, 0 \mid \text{Suc} \, _\Rightarrow \, 0)
\]

Nested patterns:

\[
(case \, xs \, of \, [0] \Rightarrow 0 \mid [\text{Suc} \, \, n] \Rightarrow \, n \mid \, _\Rightarrow \, 2)
\]

Complicated patterns mean complicated proofs!

Need ( ) in context
Tree_Demo.thy
3 Type and function definitions

Type definitions

Function definitions
Non-recursive definitions

Example:

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

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

How about \( f(x) = f(x) + 1 \)?

\[ \Rightarrow 0 = 1 \]

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

• Pattern-matching over datatype constructors

• Order of equations matters

• Termination must be provable automatically by size measures

• Proves customized induction schema
Example: separation

\[
\text{fun} \ sep :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list \ \text{where} \\
sep\ a\ (x\#y\#zs) = x \# a \# \ sep\ a\ (y\#zs) \ | \\
sep\ a\ xs = xs
\]
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:
- (Suc m, 0) > (m, Suc 0)
- (Suc m, Suc n) > (Suc m, n)
- (Suc m, Suc n) > (m, _)

77
A restrictive version of \texttt{fun}

Means \textit{primitive recursive}

Most functions are primitive recursive

Frequently found in Isabelle theories

The essence of primitive recursion:

\[
\begin{align*}
  f(0) &= \ldots \quad \text{no recursion} \\
  f(Suc\ n) &= \ldots f(n) \ldots \\
  g([]) &= \ldots \quad \text{no recursion} \\
  g(x#xs) &= \ldots g(xs) \ldots
\end{align*}
\]
2 Overview of Isabelle/HOL
3 Type and function definitions
4 Induction and Simplification
5 Case Study: IMP Expressions
6 Logic and Proof beyond “=”
7 Isar: A Language for Structured Proofs
4 Induction and Simplification

Induction

Simplification
Basic induction heuristics

Theorems about recursive functions are proved by induction

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

Our initial reverse:

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

A tail recursive version:

```scala
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

\textbf{fun} \texttt{div2 :: nat} → \texttt{nat where}

\texttt{div2 0} = \texttt{0} \mid 
\texttt{div2 (Suc 0)} = \texttt{0} \mid 
\texttt{div2 (Suc(Suc n))} = \texttt{Suc(div2 n)}

\textbf{induction rule div2.induct:}

\begin{align*}
P(0) \quad P(Suc 0) \quad \land n. \quad P(n) \implies P(Suc(Suc n))
\end{align*}

\[ P(m) \]
Computation Induction

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

\begin{align*}
\text{for each defining equation} \\
\quad f(e) &= \ldots f(r_1) \ldots f(r_k) \ldots \\
\text{prove } P(e) \text{ assuming } P(r_1), \ldots, P(r_k).
\end{align*}

Induction follows course of (terminating!) computation

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

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

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

Heuristic:

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

Computation Induction
4 Induction and Simplification

Induction

Simplification
Simplification means . . .

Using equations $l = r$ from left to right
As long as possible

Terminology: equation $\sim$ *simplification rule*

Simplification $= (\text{Term})$ Rewriting
An example

Equations:

\[0 + n = n\] (1)

\[(Suc \ m) + n = Suc (m + n)\] (2)

\[(Suc \ m \leq Suc \ n) = (m \leq n)\] (3)

\[(0 \leq m) = True\] (4)

Rewriting:

\[0 + Suc \ 0 \leq Suc \ 0 + x\] (1) \[\equiv\]

\[Suc \ 0 \leq Suc \ 0 + x\] (2) \[\equiv\]

\[Suc \ 0 \leq Suc (0 + x)\] (3) \[\equiv\]

\[0 \leq 0 + x\] (4) \[\equiv\]

True
Conditional rewriting

Simplification rules can be conditional:

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

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

Example:

\[
p(0) = \text{True}
\]

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

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 \textit{simp}-rules (almost) blindly from left to right.

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

\[
\begin{array}{l}
\left[ P_1; \ldots; P_k \right] \implies l = r \\
\end{array}
\]

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

\[
\begin{array}{l}
n < m \implies (n < \text{Suc} \ m) = True \quad \text{YES} \\
\text{Suc} \ n < m \implies (n < m) = True \quad \text{NO}
\end{array}
\]
Proof method \textit{simp}

Goal: 1. \([ P_1; \ldots; P_m ] \Rightarrow C\)

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

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

\begin{itemize}
  \item lemmas with attribute \textit{simp}
  \item rules from \textbf{fun} and \textbf{datatype}
  \item additional lemmas \(eq_1 \ldots eq_n\)
  \item assumptions \(P_1 \ldots P_m\)
\end{itemize}

Variations:

\begin{itemize}
  \item \((simp \ldots \ del: \ldots)\) removes \textit{simp}-lemmas
  \item \textit{add} and \textit{del} are optional
\end{itemize}
auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
- *auto* can also be modified:
  
  (auto simp add: ... simp del: ... )
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 \texttt{auto}. Similar for any datatype \texttt{t}: \texttt{t.split}
Simp_Demo.thy
2 Overview of Isabelle/HOL
3 Type and function definitions
4 Induction and Simplification
5 Case Study: IMP Expressions
6 Logic and Proof beyond “=”
7 Isar: A Language for Structured Proofs
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

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

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** \( vname = \text{string} \)

**datatype** \( aexp = N \text{ int} | V vname | \text{Plus} aexp aexp \)

<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>( \text{Plus} (V &quot;x&quot;) (V &quot;y&quot;) )</td>
</tr>
<tr>
<td>2+(z+3)</td>
<td>( \text{Plus} (N 2) (\text{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 *state*.
- The state is a function from variable names to values:

  ```
  type_synonym val = int
  type_synonym state = vname ⇒ val
  ```
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
5 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
ASM.thy
This was easy. Because evaluation of expressions always terminates. But execution of programs may \textit{not} terminate. Hence we cannot define it by a total recursive function.

\begin{quote}
We need more logical machinery to define program execution and reason about it.
\end{quote}
2 Overview of Isabelle/HOL

3 Type and function definitions

4 Induction and Simplification

5 Case Study: IMP Expressions

6 Logic and Proof beyond “=”

7 Isar: A Language for Structured Proofs
6 Logic and Proof beyond “=”

Logical Formulas
Proof Automation
Single Step Proofs
Inductive Definitions
Syntax (in decreasing precedence):

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

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 \(\to\))
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)

\[ P \land \forall x. Q x \sim P \land (\forall x. Q x) \]
X-Symbols

... and their ascii representations:

\[ \forall \quad \text{\textbackslash<forall>} \quad \text{ALL} \]
\[ \exists \quad \text{\textbackslash<exists>} \quad \text{EX} \]
\[ \lambda \quad \text{\textbackslash<lambda>} \quad \% \]
\[ \rightarrow \quad --\rightarrow \]
\[ \leftrightarrow \quad <---\rightarrow \]
\[ \land \quad \\land \quad \& \]
\[ \lor \quad \| \quad | \]
\[ \neg \quad \text{\textbackslash<not>} \quad \sim \]
\[ \neq \quad \text{\textbackslash<noteq>} \quad \sim= \]
Sets over type 'a

'a set = 'a \Rightarrow bool

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

\in \ (<\text{in}>), \subseteq \ (<\text{subseteq}>), \leq,
\cup \ (<\text{union}>), Un
\cap \ (<\text{inter}>), 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 variables in \( t \).
Logic and Proof beyond “=”

Logical Formulas
Proof Automation
Single Step Proofs
Inductive Definitions
simp and auto

**simp**: rewriting and a bit of arithmetic

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

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

Exception: auto acts on all subgoals
• rewriting, logic, sets, relations and a bit of arithmetic.
• **incomplete** but better than *auto*.
• Succeeds or fails
• Extensible with new *simp*-rules
blast

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

\texttt{arith}:

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

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

Do you feel lucky?

---

1Automatic Theorem Provers
by \((proof-method)\)

\[\approx\]

apply \((proof-method)\)
done
Auto_Proof_Demo.thy
6 Logic and Proof beyond “=”

Logical Formulas
Proof Automation
Single Step Proofs
Inductive Definitions
Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed by taking the goal apart.
What are these ?-variables ?

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

Example: theorem conjI: $[?P; ?Q] \implies ?P \land ?Q$

These ?-variables can later be instantiated:

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

- 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 \( A_1 \ldots A_n \)

\textbf{apply} (\textit{rule xyz})

“Backchaining”
Typical backwards rules

\[
\frac{?P \quad ?Q}{?P \land ?Q} \quad \text{conjI}
\]

\[
\frac{?P \iff ?Q}{?P \implies ?Q} \quad \text{impI} \quad \frac{\forall x. ?P x}{\land x. ?P x} \quad \text{allI}
\]

\[
\frac{?P \iff ?Q \quad ?Q \iff ?P}{?P = ?Q} \quad \text{iffI}
\]

They are known as introduction rules because they introduce a particular connective.
Teaching *blast* new intro rules

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

$$(blast \ intro: \ r)$$

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

Example:

**Theorem** $trans$: $[ ?x \leq ?y; \ ?y \leq \ ?z ] \Rightarrow \ ?x \leq \ ?z$

**Goal** 1. $[ a \leq b; \ b \leq c; \ c \leq d ] \Rightarrow a \leq d$

**Proof** $apply(blast \ intro: \ trans)$

Can greatly increase the search space!
Forward proof: OF

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

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

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

Example: theorem \texttt{refl}: \( \mathcal{t} = \mathcal{t} \)

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

\[ \rightsquigarrow \]

\( a = a \land b = b \)
From now on: ? mostly suppressed on slides
SingleStepDemo.thy
versus

is part of the Isabelle framework. It structures theorems and proof states: \([ A_1; \ldots; A_n ] \implies A\)

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

Phrase theorems like this \([ A_1; \ldots; A_n ] \implies A\) not like this \(A_1 \land \ldots \land A_n \implies A\)
Logic and Proof beyond “=”

- Logical Formulas
- Proof Automation
- Single Step Proofs
- Inductive Definitions
Example: even numbers

Informally:

- 0 is even
- If \( n \) is even, so is \( n + 2 \)
- These are the only even numbers

In Isabelle/HOL:

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

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

```haskell
fun even :: nat ⇒ bool where
even 0 = True |
even (Suc 0) = False |
even (Suc (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 \texttt{ev.induct}:

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

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

\[
\begin{align*}
& \left[ I \ a_1; \ldots ; I \ a_n \right] \Rightarrow I \ a \\
& \vdots
\end{align*}
\]

**Note:**
- $I$ may have multiple arguments.
- Each rule may also contain *side conditions* not involving $I$. 
Rule induction in general

To prove

\[ I \ x \ \Rightarrow \ P \ x \]

by *rule induction* on \( I \ x \)

we must prove for every rule

\[
\begin{array}{c}
\left[ I \ a_1; \ldots ; I \ a_n \right] \ \Rightarrow \ I \ a
\end{array}
\]

that \( P \) is preserved:

\[
\begin{array}{c}
\left[ I \ a_1; P \ a_1; \ldots ; I \ a_n; P \ a_n \right] \ \Rightarrow \ P \ a
\end{array}
\]
Rule induction is absolutely central to (operational) semantics and the rest of this lecture course
Inductive_Demo.thy
Inductively defined sets

\texttt{inductive\_set \: I :: \tau \ \textit{set} \ \textbf{where}}

\[ \begin{array}{c}
\left[ a_1 \in I; \ldots; a_n \in I \right] \implies a \in I
\end{array} \]

Difference to \texttt{inductive}:\n
- arguments of \( I \) are tupled, not curried
- \( I \) can later be used with set theoretic operators, eg \( I \cup \ldots \)
2 Overview of Isabelle/HOL
3 Type and function definitions
4 Induction and Simplification
5 Case Study: IMP Expressions
6 Logic and Proof beyond “=”
7 Isar: A Language for Structured Proofs
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 (\implies)
   | [from fact^+] (have | show) prop proof

prop = [name:] ”formula”

fact = name | ...
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
Example: Cantor’s theorem

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

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

\[(\text{have}|\text{show}) \ \text{prop} \ using \ \text{facts} \]

\[= \]

\[\text{from facts} \ (\text{have}|\text{show}) \ \text{prop} \]

\[= \]

\[\text{with facts} \]

\[= \]

\[\text{from facts} \ this\]
Structured lemma statement

**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: A Language for Structured Proofs

Isar by example

Proof patterns

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks

Induction

Rule Induction

Rule Inversion
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
Contradiction

show \( \neg P \)
proof
  assume \( P \)
  :
  show \( False \) \ldots
qed

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

proof

assume $P$

\[\vdots\]

show $Q$ \ldots

next

assume $Q$

\[\vdots\]

show $P$ \ldots

qed
\( \forall \) and \( \exists \) introduction

\begin{align*}
& \text{show } \forall x. \ P(x) \\
& \text{proof} \\
& \quad \text{fix } x \quad \text{local fixed variable} \\
& \quad \text{show } P(x) \ldots \\
& \text{qed}
\end{align*}

\begin{align*}
& \text{show } \exists x. \ P(x) \\
& \text{proof} \\
& \quad : \\
& \quad \quad \text{show } P(\text{witness}) \ldots \\
& \text{qed}
\end{align*}
∃ 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 \neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set})

proof
  assume \text{surj } f
  hence \exists a. \{x. x \notin f \ x\} = f \ a \ \text{by (auto simp: surj_def)}
  then obtain a where \{x. x \notin f \ x\} = f \ a \ \text{by blast}
  hence a \notin f \ a \ \leftrightarrow \ a \in f \ a \ \text{by blast}
  thus False \ \text{by blast}

qed
Set equality and subset

\[
\text{show } A = B \\
\text{proof} \\
\quad \text{show } A \subseteq B \ldots \\
\text{next} \\
\quad \text{show } B \subseteq A \ldots \\
\text{qed}
\]

\[
\text{show } A \subseteq B \\
\text{proof} \\
\quad \text{fix } x \\
\quad \text{assume } x \in A \\
\quad : \\
\quad \text{show } x \in B \ldots \\
\text{qed}
\]
Isar_Demo.thy

Exercise
Isar: A Language for Structured Proofs

Isar by example
Proof patterns

Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
Example: pattern matching

show \( formula_1 \leftrightarrow formula_2 \) (is \(?L \leftrightarrow ?R\))

proof

assume \(?L\)

: show \(?R\) ... 

next

assume \(?R\)

: show \(?L\) ... 

qed
show formula (is thesis)
proof -
| show thesis ... |
qed

Every show implicitly defines thesis
Introducing local abbreviations in proofs:

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

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

By name:

```
have x0: "x > 0" . . .
:
from x0 . . .
```

By value:

```
have "x > 0" . . .
:
from 'x>0' . . .

↑ ↑

back quotes
```
Isar_Demo.thy

Pattern matching and quotation
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
Example

**Lemma**

**Assumes** $xs = \text{rev } xs$

**Shows** $(\exists ys. \; xs \equiv ys \circ \text{rev } ys) \lor$

$(\exists ys \; a. \; xs \equiv ys \circ a \neq \text{rev } ys)$

**Proof** ???
Isar_Demo.thy

Top down proof development
When automation fails

Split proof up into smaller steps.

Or explore by apply:

have . . . using . . .
apply -
apply auto
apply . . .

At the end:

• done
• Better: convert to structured proof
7 Isar: A Language for Structured Proofs
Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
moreover—ultimately

have $P_1 \ldots$
moreover
have $P_2 \ldots$
moreover
have $P_n \ldots$
ultimately
have $P \ldots$

have $\text{lab}_1: P_1 \ldots$
have $\text{lab}_2: P_2 \ldots$

$\approx$

have $\text{lab}_n: P_n \ldots$
from $\text{lab}_1 \text{lab}_2 \ldots$

have $P \ldots$

With names
\[
\begin{array}{l}
\{ \text{fix } x_1 \ldots x_n \\
\text{assume } A_1 \ldots A_m \\
\vdots \\
\text{have } B \\
\} \\
\end{array}
\]
proves \([A_1; \ldots; A_m] \implies B\)
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):

\[ \forall x_1 \ldots x_n \left[ A_1; \ldots ; A_m \right] \implies B \]

How to prove each subgoal:

\begin{itemize}
\item \textbf{fix} \( x_1 \ldots x_n \)
\item \textbf{assume} \( A_1 \ldots A_m \)
\item \textbf{show} \( B \)
\end{itemize}

Separated by \textbf{next}
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
Isar_Induction_Demo.thy

Case distinction
Datatype case distinction

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

proof \((cases "term")\)

\[
\begin{align*}
\text{case } (C_1 x_1 \ldots x_k) \\
\ldots x_j \ldots
\end{align*}
\]

next

\[
\vdots
\]

qed

where \( \text{case } (C_i x_1 \ldots x_k) \equiv \)

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

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

\begin{align*}
\text{show } & P(n) \\
\text{proof } & (\text{induction } n) \\
\quad \text{case } & 0 \\
\quad & \vdots \\
\quad \text{show } & ?\text{case} \\
\text{next } & \\
\quad \text{case } & (\text{Suc } n) \\
\quad & \vdots \\
\quad & \vdots \\
\quad \text{show } & ?\text{case} \\
\text{qed}
\end{align*}

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

show $A(n) \Rightarrow P(n)$

proof ($\text{induction } n$)

\begin{align*}
\text{case } 0 & \quad \equiv \quad \text{assume } 0: A(0) \\
\text{let } ?\text{case} = P(0) \\
\text{show } ?\text{case} & \quad \equiv \quad \text{fix } n \\
\text{assume } Suc: A(n) \Rightarrow P(n) \\
& \quad \quad \quad A(Suc \ n) \\
& \quad \quad \quad \text{let } ?\text{case} = P(Suc \ n)
\end{align*}

next

\text{case } (\text{Suc } n) \\
\text{show } ?\text{case} \\
\text{qed}
Named assumptions

In a proof of

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

by structural induction:
In the context of

**case** \( C \)

we have

- **C.IH** the induction hypotheses
- **C.prems** the premises \( A_i \)

\[ C \quad C.IH + C.prems \]
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. \(?\text{case}\)
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
Isar_Induction_Demo.thy

Rule induction
Rule induction

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

where

$\text{rule}_1: \ldots$

$\vdots$

$\text{rule}_n: \ldots$

**show** $I \ x \ y \Rightarrow P \ x \ y$

**proof** *(induction rule: $I$.induct)*

**case** $\text{rule}_1$

$\ldots$

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

next

$\vdots$

$\vdots$

**case** $\text{rule}_n$

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

qed
Fixing your own variable names

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

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

In a proof of

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

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

In the context of

**case** \( R \)

we have

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

\( R \quad R.IH + R.hyps + R.prems \)
7 Isar: A Language for Structured Proofs
Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Rule Inversion
Rule inversion

\textbf{inductive} \ ev :: \ \textit{nat} \Rightarrow \ \textit{bool} \ \textbf{where}

\begin{align*}
ev\partial &: \ ev \ 0 \\
evSS &: \ ev \ n \Rightarrow ev(Suc(Suc \ n))
\end{align*}

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

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

\textbf{Rule inversion} = case distinction over rules
Isar_Induction_Demo.thy

Rule inversion
from `ev n` have $P$

proof cases

  case `ev0`

  : show $?thesis$ ...

next

  case `(evSSS k)`

  : show $?thesis$ ...

qed

Impossible cases disappear automatically
Part II

IMP: A Simple Imperative Language
8 IMP

9 Compiler

10 A Typed Version of IMP
8 IMP

9 Compiler

10 A Typed Version of IMP
Terminology

**Statement:** declaration of fact or claim

*Semantics is easy.*

**Command:** order to do something

*Study the book until you have understood it.*

Expressions are *evaluated*, commands are *executed*
Concrete syntax:

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

Abstract syntax:

\[
\text{datatype} \ com \ = \ SKIP \\
| \ Assign \ string \ aexp \\
| \ Semi \ com \ com \\
| \ If \ bexp \ com \ com \\
| \ While \ bexp \ com
\]
Com.thy
8 IMP

Big Step Semantics

Small Step Semantics
Big step semantics

Concrete syntax:

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

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

“⇒” here not type!
Big step rules

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

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

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

\[(c_1; c_2, s_1) \Rightarrow s_3\]
Big step rules

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

\[
\frac{\neg\ bval\ b\ s\ \ (c_2,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}
\]
Big step rules

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

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

where

\[
egin{align*}
\begin{aligned}
\text{where} & \quad w & = & \text{WHILE } b \text{ DO } c \\
b & = & \text{NotEq} (V "x") (N 2) \\
c & = & "x" ::= \text{Plus} (V "x") (N 1) \\
s_i & = & s("x" ::= i)
\end{aligned}
\end{align*}
\]

\[
\begin{align*}
\text{NotEq } a_1 a_2 & = \\
\text{Not(And (Not(Less } a_1 a_2)) (Not(Less } a_2 a_1)))
\end{align*}
\]
Logically speaking

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

is just infix syntax for

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

where

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

is an inductively defined predicate.
Big_STEP.thy

Semantics
Rule inversion

What can we deduce from

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

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

Replaces assm \((c_1; c_2, s_1) \Rightarrow s_3\) by two assms \((c_1, s_1) \Rightarrow s_2\) and \((c_2, s_2) \Rightarrow s_3\) (with a new fixed \(s_2\)). No \(\exists\) and \(\land\)!
The general format: elimination rules

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

\[
\therefore P
\]

(possibly with $\bigwedge \overline{x}$ in front of the $\text{asm}_i \implies P$)

Reading:

To prove a goal $P$ with assumption $\text{asm}$, prove all $\text{asm}_i \implies P$

Example:

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

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

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

Rule inversion
Command equivalence

Two commands have the same input/output behaviour:

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

where $w = WHILE b DO c$

$iw = IF b THEN c; w ELSE SKIP$

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

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

\[\iff\]

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

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

\[\iff\]

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

Using the rules and rule inversions for \(\Rightarrow\).
Big_Step.thy

Command equivalence
Execution is deterministic

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

\[(c, s) \Rightarrow t \implies (c, s) \Rightarrow t' \implies t = t'\]

Proof by rule induction, for arbitrary \(t'\).
Big_Step.thy

Execution is deterministic
The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

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

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

- nonterminating computations,
- parallel computations.

We need a finer grained semantics!
8 IMP

Big Step Semantics

Small Step Semantics
Small step semantics

Concrete syntax:

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

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:

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

Why?

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

\[
\begin{align*}
(x := a, s) & \rightarrow (\text{SKIP}, s(x := \text{aval} a s)) \\
(\text{SKIP}; c, s) & \rightarrow (c, s) \\
(c_1, s) & \rightarrow (c'_1, s') \\
(c_1; c_2, s) & \rightarrow (c'_1; c_2, s')
\end{align*}
\]
Small step rules

\[ \text{bval} \ b \ s \]

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

\[ \neg \ \text{bval} \ b \ s \]

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

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

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

\[
(\text{"z"} ::= V \text{"x"}; \text{"x"} ::= V \text{"y"}; \text{"y"} ::= V \text{"z"}, s) \rightarrow \ldots
\]

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

\[
(w, s_0) \rightarrow \ldots
\]

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

\[
b = \text{Less (V \text{"x"}) (N 1)}
\]

\[
c = \text{"x" ::= Plus (V \text{"x"}) (N 1)}
\]

\[
s_n = \langle \text{"x"} := n \rangle
\]
Small_Step.thy

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

Proof by rule induction (of course on \( cs \Rightarrow t \))
From $\rightarrow^*$ to $\Rightarrow$

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

Needs to be generalized:

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

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

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

**Needs**

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

Lemma 2 is proved by rule induction on $cs \rightarrow cs'$.
Corollary $cs \Rightarrow t \iff cs \rightarrow^* (SKIP, t)$
Small_Step.thy

Equivalence of big and small
Can execution stop prematurely?

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

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

We prove the contrapositive

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

by induction on \(c\).

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

Together:

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

⇒ yields final state iff → terminates

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

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

\[= \hspace{0.5cm} (\exists t. \ cs \rightarrow^* (\text{SKIP}, t))\]

(by big = small)

\[= \hspace{0.5cm} (\exists cs'. \ cs \rightarrow^* cs' \land \text{final} \ cs')\]

(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 \( cs \rightarrow cs' \rightarrow \ldots \).
8 IMP

9 Compiler

10 A Typed Version of IMP
Compiler

Stack Machine

Compiler
Stack Machine

Instructions:

**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 `≥`
Semantics

Type synonyms:

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

Execution of 1 instruction:

\[ \text{instr} \vdash i \ (pc, s, stk) \rightarrow (pc', s', stk') \]
\[ \text{instr} \vdash i \ \text{config} \rightarrow \text{config} \]
Single Instructions

\textit{LOADI} \ n
\[
\vdash i \ (i, \ s, \ stk) \rightarrow (i + 1, \ s, \ n \ # \ stk)
\]

\textit{LOAD} \ x
\[
\vdash i \ (i, \ s, \ stk) \rightarrow (i + 1, \ s, \ s \ x \ # \ stk)
\]

\textit{ADD}
\[
\vdash i \ (i, \ s, \ stk) \rightarrow (i + 1, \ s, \ (hd2 \ stk + hd \ stk) \ # \ tl2 \ stk)
\]

\textit{STORE} \ x \vdash i \ (i, \ s, \ stk) \rightarrow (i + 1, \ s(x := hd \ stk), \ tl \ stk)
Single Instructions

\[ \text{JMP } n \vdash i \ (i, \ s, \ stk) \rightarrow (i + 1 + n, \ s, \ stk) \]

\[ \text{JMPIE \ LESS } n \]
\[ \vdash i \ (i, \ s, \ stk) \rightarrow \]
\[ (\text{if } \text{hd} \text{2 stk} < \text{hd} \text{ stk} \text{ then } i + 1 + n \text{ else } i + 1, \ s, \]
\[ \text{tl} \text{2 stk}) \]

\[ \text{JMPIE \ GE } n \]
\[ \vdash i \ (i, \ s, \ stk) \rightarrow \]
\[ (\text{if } \text{hd} \text{ stk} \leq \text{hd} \text{2 stk} \text{ then } i + 1 + n \text{ else } i + 1, \ s, \]
\[ \text{tl} \text{2 stk}) \]
Lifting to Programs

Programs are instruction lists.

Executing one program step:

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

\( instr\ list \vdash config \rightarrow config \)

\[ P \vdash c \rightarrow c' = \exists i\ s\ stk. \]

\[ c = (i, s, stk) \land \]

\[ P \land i \vdash (i, s, stk) \rightarrow c' \land 0 \leq i \land i < isize\ P \]

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

Defined in the usual manner:

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

Stack Machine
Compiler

Stack Machine

Compiler
Compiling $aexp$

Same as before:

\[
\begin{align*}
\text{acomp } (N \ n) &= \text{[LOADI } n] \\
\text{acomp } (V \ x) &= \text{[LOAD } x] \\
\text{acomp } (\text{Plus } a1 \ a2) &= \text{acomp } a1 @ \text{acomp } a2 @ [\text{ADD}]
\end{align*}
\]

Correctness theorem:

\[
\text{acomp } a \\
\vdash (0, s, stk) \rightarrow^* (\text{isize } (\text{acomp } a), s, \text{aval } a s \# stk)
\]

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

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

\[ P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies P'@P \]
\[ \vdash (\text{isize } P' + i, s, stk) \rightarrow^* (\text{isize } P' + i', s', stk') \]

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

\[ P \vdash c \rightarrow c' \implies P@P' \vdash c \rightarrow c' \]

\[ P \vdash (i, s, stk) \rightarrow (i', s', stk') \implies P'@P \vdash (\text{isize } P' + i, s, stk) \rightarrow (\text{isize } P' + i', s', stk') \]

Proofs by cases/induction.
Compiling \textit{bexp}

Let \textit{ins} be the compilation of \textit{b}:

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

Principle:

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

Parameters: \textbf{when} to jump (if \textit{b} is \textit{True} or \textit{False})
\textbf{where} to jump to (\textit{n})

\textit{bcomp :: bexp }\Rightarrow\text{ bool }\Rightarrow\text{ int }\Rightarrow\text{ instr list}
Example

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

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

\[
\text{[LOAD "x", LOAD "y", LOAD "z", LOAD "a",]}
\]
\[
\text{bcomp :: bexp} \Rightarrow \text{bool} \Rightarrow \text{int} \Rightarrow \text{instr list}
\]
\[
bcomp \ (Bc \ v) \ c \ n = (\text{if } v = c \ \text{then } [\text{JMP } n] \ \text{else } [])
\]
\[
bcomp \ (\text{Not } b) \ c \ n = bcomp b \ (\neg c) \ n
\]
\[
bcomp \ (\text{Less } a1 \ a2) \ c \ n =
\]
\[
acomp a1 \ @
\]
\[
acomp a2 \ @ (\text{if } c \ \text{then } [\text{JMPLESS } n] \ \text{else } [\text{JMPGE } n])
\]
\[
bcomp \ (\text{And } b1 \ b2) \ c \ n =
\]
\[
\text{let } cb2 = bcomp b2 c n;
\]
\[
m = \text{if } c \ \text{then isize } cb2 \ \text{else isize } cb2 + n;
\]
\[
\text{cb1 = bcomp b1 False m}
\]
\[
in \ cb1 \ @ \ cb2
\]
Correctness of \( \text{bcomp} \)

\[
0 \leq n \implies \text{bcomp } b \ c \ n
\]

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

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

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

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

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

\[
\text{let } cc_1 = \text{ccomp } c_1; \ cc_2 = \text{ccomp } c_2;
\]

\[
\text{cb = bcomp } b \ False \ (\text{isize } cc_1 + 1)
\]

\[
\text{in } cb @ cc_1 @ \text{JMP (isize } cc_2 \ # \ cc_2
\]

\[
\text{ccomp \ (WHILE \ b \ DO \ c) =}
\]

\[
\text{let } cc = \text{ccomp } c;
\]

\[
\text{cb = bcomp } b \ False \ (\text{isize } cc + 1)
\]

\[
\text{in } cb @ cc @ [\text{JMP } (\text{isize } cb + \text{isize } cc + 1)]
\]
Correctness of \textit{ccomp}

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

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

Proof by rule induction.
The other direction

We have only shown “⇒”: 

*compiled code simulates source code.*

How about “⇐”:

*source code simulates compiled code?*

If \( c \text{comp} \ c \) with start state \( s \) produces result \( t \), and if(!) \((c, s) \Rightarrow t'\), then “⇒” 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.
The other direction

Two approaches:

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

- A direct proof:
  IMP/Comp_Rev.thy in the Isabelle distribution.
A Typed Version of IMP
A Typed Version of IMP

Remarks on Type Systems

Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed IMP
Why Types?

To prevent mistakes, dummy!
There are 3 kinds of types

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

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

**The Ugly**  Dynamic types that detect errors when it can be too late.
Example: “TypeError: . . .” in Python.
The ideal

Well-typed programs cannot go wrong.


The most influential slogan and one of the most influential papers in programming language theory.
What could go wrong?

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

There are type systems for *everything* (and more) but in practice (Java, C#) only 1 is covered.
A programming language is type safe if the execution of a well-typed program cannot lead to certain errors.

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

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

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

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

How about **completeness**? Remember Rice:

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

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

Values:

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

The state:

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

Arithmetic expressions:

\textbf{datatype} \ \aexp =
\begin{align*}
\textit{Ic} \ \textit{int} \ | \ \textit{Rc} \ \textit{real} \ | \ \textit{V} \ \textit{vname} \ | \ \text{Plus} \ \aexp \ \aexp
\end{align*}
Why tagged values?

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

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

Not recursive function but inductive predicate:

\[
\begin{align*}
taval :: aexp & \Rightarrow state \Rightarrow val \Rightarrow bool \\
taval (Ic \ i) & s (Iv \ i) \\
taval (Rc \ r) & s (Rv \ r) \\
taval (V \ x) & s (s \ x) \\
\frac{\text{taval } a_1 \ s \ (Iv \ i_1) \quad \text{taval } a_2 \ s \ (Iv \ i_2)}{\text{taval } (Plus \ a_1 \ a_2) \ s \ (Iv \ (i_1 + i_2))} \\
\frac{\text{taval } a_1 \ s \ (Rv \ r_1) \quad \text{taval } a_2 \ s \ (Rv \ r_2)}{\text{taval } (Plus \ a_1 \ a_2) \ s \ (Rv \ (r_1 + r_2))}
\end{align*}
\]
Example: evaluation of $Plus\ (V\ "x")\ (Ic\ 1)$

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

$$taval\ (V\ "x")\ s\ (Iv\ i)\ taval\ (Ic\ 1)\ s\ (Iv\ 1)$$

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

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

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

An extremely useful datatype:

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

A “partial” function:

```
taval :: aexp ⇒ state ⇒ val option
```

Exercise!
Boolean expressions

Syntax as before. Semantics:

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

\[
\text{tbval} \ (Bc \ v) \ s \ v \quad \frac{\text{tbval} \ b \ s \ bv}{\text{tbval} \ (\text{Not} \ b) \ s \ (\neg \ bv)}
\]

\[
\text{tbval} \ b_1 \ s \ bv_1 \quad \text{tbval} \ b_2 \ s \ bv_2
\]

\[
\frac{\text{tbval} \ (\text{And} \ b_1 \ b_2) \ s \ (bv_1 \land bv_2)}{}
\]

\[
\text{taval} \ a_1 \ s \ (Iv \ i_1) \quad \text{taval} \ a_2 \ s \ (Iv \ i_2)
\]

\[
\frac{\text{tbval} \ (\text{Less} \ a_1 \ a_2) \ s \ (i_1 < i_2)}{}
\]

\[
\text{taval} \ a_1 \ s \ (Rv \ r_1) \quad \text{taval} \ a_2 \ s \ (Rv \ r_2)
\]

\[
\frac{\text{tbval} \ (\text{Less} \ a_1 \ a_2) \ s \ (r_1 < r_2)}{}
\]
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:**
  \[
  \text{error} = \text{semantics gets stuck}
  \]
Small step semantics

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

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

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

The other rules remain unchanged.
Example

Let \( c = ("x" ::= Plus (V "x") (Ic 1)). \)

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

- If \( s "x" = Rv r : \)
  \( (c, s) \not\rightarrow \)
10 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 \( V''x'' \) and \( V''y'' \)!

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

\( tyenv = vname \Rightarrow ty \)

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

\[ \Gamma \vdash e : \tau \]
The type of an $aexp$

$$\Gamma \vdash a : \tau$$

$$tyenv \vdash aexp : ty$$

The rules:

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

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

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

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

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

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

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

Notation:

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

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

\begin{align*}
\Gamma \vdash Bc \, v \\
\Gamma \vdash b \\
\Gamma \vdash Not\, b \\
\Gamma \vdash b_1 \quad \Gamma \vdash b_2 \\
\Gamma \vdash And\, b_1 \, b_2 \\
\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau \\
\Gamma \vdash Less\, a_1 \, a_2
\end{align*}

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

Notation:

\[ \Gamma \vdash c \]
\[ tyenv \vdash com \]

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

$$
\frac{}{\Gamma \vdash \text{SKIP}} \\
\frac{\Gamma \vdash a : \Gamma \ x}{\Gamma \vdash x ::= a} \\
\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \\
\frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2} \\
\frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash \text{WHILE } b \text{ DO } c}
$$
Syntax-directedness

All three sets of typing rules are syntax-directed:

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

Therefore each set of rules is executable without backtracking:

Given \( \Gamma \) and a term \( a/b/c \), its well-typedness (and its type) is computable by backchaining without backtracking.

The big and small step semantics are not syntax-directed.
All three sets of typing rules are **compositional**: 

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

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

The big step semantics is not compositional because the execution of *WHILE* depends on the execution of *WHILE*. 
A Typed Version of IMP
Remarks on Type Systems
Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed IMP
Well-typed states

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

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

The state must be well-typed w.r.t. $\Gamma$.

Frequent alternative terminology:
The state must conform to $\Gamma$. 

The type of a value:

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

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:

\[
\begin{align*}
\llbracket \Gamma \vdash c; \Gamma \vdash s; c \neq SKIP \rrbracket & \implies \exists cs'. (c, s) \to cs' \\
\end{align*}
\]

Preservation:

\[
\begin{align*}
\llbracket (c, s) \to (c', s'); \Gamma \vdash c; \Gamma \vdash s \rrbracket & \implies \Gamma \vdash s' \\
\llbracket (c, s) \to (c', s'); \Gamma \vdash c \rrbracket & \implies \Gamma \vdash c'
\end{align*}
\]

Type soundness:

\[
\begin{align*}
\llbracket (c, s) \to^\ast (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq SKIP \rrbracket & \implies \exists cs''. (c', s') \to cs''
\end{align*}
\]
Progress:

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

\[ \frac{}{\text{aexp}} \]

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

Preservation:

\[ [\Gamma \vdash a : \tau; \text{taval} a s v; \Gamma \vdash s] \Longrightarrow \text{type} v = \tau \]
All proofs by rule induction.
Types.thy
The mantra

Type systems have a purpose:

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

The correctness of the prediction must be provable.
Part III

Data-Flow Analyses and Optimization
11 Definite Assignment Analysis

12 Live Variable Analysis

13 Information Flow Analysis
Definite Assignment Analysis

Live Variable Analysis

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

Java Language Specification

Java was the first language to force programmers to initialize their variables.
Examples: ok or not?

Assume "x" is initialized:

\[
\begin{align*}
\text{IF } \text{Less} \left( V \ "x" \right) \left( N \ 1 \right) \text{ THEN } & \quad "y" ::= V \ "x" \\
\text{ELSE } & \quad "y" ::= \text{Plus} \left( V \ "x" \right) \left( N \ 1 \right) \\
\end{align*}
\]

Assume "x" and "y" are initialized:

\[
\begin{align*}
\text{WHILE } \text{Less} \left( V \ "x" \right) \left( V \ "y" \right) \text{ DO } & \quad "z" ::= V \ "x"; \\
& \quad "z" ::= \text{Plus} \left( V \ "z" \right) \left( N \ 1 \right) \\
\end{align*}
\]
Simplifying principle

We do not analyze boolean expressions to determine program execution.
Definite Assignment Analysis

Prelude: Variables in Expressions

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

$$
\text{vars :: 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 :: 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
$$
Vars.thy
Definite Assignment Analysis

Prelude: Variables in Expressions

Definite Assignment Analysis

Initialization Sensitive Semantics
Modified example from the JLS:

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

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

\[ D \ A \ c \ A' \] 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.
\[
D A \SKIP A \\
\text{\textit{vars} } a \subseteq A \\
D A (x ::= a) \ (\text{insert } x \ A) \\
D A_1 \ c_1 \ A_2 \quad D A_2 \ c_2 \ A_3 \\
D A_1 \ (c_1; \ c_2) \ A_3 \\
\text{\textit{vars} } b \subseteq A \quad D A \ c_1 \ A_1 \quad D A \ c_2 \ A_2 \\
D A \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ (A_1 \cap A_2) \\
\text{\textit{vars} } b \subseteq A \quad D A \ c \ A' \\
D A \ (\text{WHILE } b \ \text{DO } c) \ A
\]
Correctness of $D$

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

- Small step semantics: semantics shorter, correctness proof longer

For variety’s sake, we choose a big step semantics.
Definite Assignment Analysis

Prelude: Variables in Expressions

Definite Assignment Analysis

Initialization Sensitive Semantics
\[ \text{state} = \text{vname} \Rightarrow \text{val option} \]

where

**datatype** \('a\) option = None \(\mid\) Some \('a\)

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

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

\[
\begin{align*}
\text{aval} &:: \text{aexp} \Rightarrow \text{state} \Rightarrow \text{val option} \\
\text{aval} (N \ i) \ s &= \text{Some} \ i \\
\text{aval} (V \ x) \ s &= s \ x \\
\text{aval} (\text{Plus} \ a_1 \ a_2) \ s &= \\
\quad (\text{case} \ (\text{aval} \ a_1 \ s, \ \text{aval} \ a_2 \ s) \ \text{of} \\
\quad \quad (\text{Some} \ i_1, \ \text{Some} \ i_2) \Rightarrow \text{Some}(i_1+i_2) \\
\quad \quad \mid - \Rightarrow \text{None})
\end{align*}
\]
\textit{bval} :: bexp \Rightarrow \text{state} \Rightarrow \text{bool option}

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

\texttt{bval \ (Not \ b) \ s =}
\begin{cases} \texttt{(case \ bval \ b \ s \ of \ None \Rightarrow \ None} \\
| \texttt{Some \ bv \Rightarrow \ Some \ (\neg \ bv)}) \end{cases}

\texttt{bval \ (And \ b_1 \ b_2) \ s =}
\begin{cases} \texttt{(case \ (bval \ b_1 \ s, \ bval \ b_2 \ s) \ of} \\
| \texttt{(Some \ bv_1, \ Some \ bv_2) \Rightarrow \ Some(bv_1 \land bv_2)} \\
| \texttt{_{-} \Rightarrow \ None)} \end{cases}

\texttt{bval \ (Less \ a_1 \ a_2) \ s =}
\begin{cases} \texttt{(case \ (aval \ a_1 \ s, \ aval \ a_2 \ s) \ of} \\
| \texttt{(Some \ i_1, \ Some \ i_2) \Rightarrow \ Some(i_1 < i_2)} \\
| \texttt{_{-} \Rightarrow \ None)} \end{cases}
Big step semantics

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

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:

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

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

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

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

\[aval \ a \ s = Some \ i\]

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

\[aval \ a \ s = None\]

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

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

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

bval b s = \text{Some False} \quad (c_2, \text{Some s}) \Rightarrow s' \\
(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \text{Some s}) \Rightarrow s' \\

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

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

\[ bval\ b\ s = None \]
\[ (\text{WHILE}\ b\ \text{DO}\ c,\ Some\ s) \Rightarrow None \]
Correctness of $D$ w.r.t. $⇒$

We want in the end:

Well-initialized programs cannot go wrong.

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

We need to prove a generalized statement:

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

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

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

Live Variable Analysis

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

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

The first assignment is redundant and can be removed because $x$ is dead at that point.
Semantically, a variable $x$ is live before command $c$ if the initial value of $x$ can influence the final state.

As a sufficient condition, we call $x$ live before $c$ if there is some potential execution of $c$ where $x$ is read before it can be overwritten. Implicitly, every variable is read at the end of $c$.

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

- $x ::= N \ 0$  😞
- $y ::= V \ x; \ y ::= N \ 0; \ x ::= N \ 0$  😊
- $\text{WHILE } b \ \text{DO } y ::= V \ x; \ x ::= N \ 1$  😊
At the end of a command, we may be interested in the value of *only some of the variables*, e.g. *only the global variables* at the end of a procedure.

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

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

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

$L\ SKIP\ X = X$

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

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

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

$vars\ b \cup L\ c_1\ X \cup L\ c_2\ X$

Example:

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

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

$L_w X$ must satisfy

- $\text{vars } b \subseteq L_w X$ (evaluation of $b$)
- $X \subseteq L_w X$ (exit)
- $L_c (L_w X) \subseteq L_w X$ (execution of $c$)
We define

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

\[ \iff \]

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

Example:

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

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

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

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

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

Proof by induction on \( c \).

\[ \implies \]

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

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

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

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

Soundness of $L$

Dead Variable Elimination

True Liveness

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

*The value of the final state on \(X\)*

*only depends on*

*the value of the initial state on \(L \cap X\).*

Put differently:

*If two initial states agree on \(L \cap X\)*

*then the corresponding final states agree on \(X\).*
Equality on

An abbreviation:

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

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

\[ s_1 = s_2 \text{ on } vars\ a \implies \text{aval\ } a\ s_1 = \text{aval\ } a\ s_2 \]
\[ s_1 = s_2 \text{ on } vars\ b \implies \text{bval\ } b\ s_1 = \text{bval\ } b\ s_2 \]
Soundness 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 $WHILE$ 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).

\[ L \ w \ X \text{ should be the least set such that } \]
\[ \text{vars } b \cup X \cup L \ c \ (L \ w \ X) \subseteq L \ w \ X. \]

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

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

Soundness of $L$

Dead Variable Elimination

True Liveness

Comparisons
Bury all assignments to dead variables:

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

\[
\text{bury \ SKIP \ X} = \ \text{SKIP}
\]

\[
\text{bury \ (x ::= a) \ X} = \ \text{if} \ x \in \ X \ \text{then} \ x ::= a \ \text{else} \ \text{SKIP}
\]

\[
\text{bury \ (c_1; c_2) \ X} = \ \text{bury} \ c_1 \ (L \ c_2 \ X); \ \text{bury} \ c_2 \ X
\]

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

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

\[(\text{bury } c \text{ UNIV}, s) \Rightarrow s' \iff (c, s) \Rightarrow s'\]

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

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

Follows from generalized statement:

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

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

Follows from generalized statement:

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

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

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

$$(bc_1;\ bc_2 = bury\ c\ X) =$$

$$\exists c_1\ c_2.\$$

$$c = c_1;\ c_2 \land$$

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

Soundness of $L$

Dead Variable Elimination

True Liveness

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

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

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

If $f \circ x \leq x$ then $x$ is a post-fixed point of $f$. 

Terminology
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 post-fixed point of

$$\lambda P. \text{vars } b \cup X \cup L^c P$$

and in particular of $L^c$. 
True liveness

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

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

Problem:

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

Better:

\[ L \left( x ::= e \right) X = \]

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

But then

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

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

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

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 \( w \)!

\( \{x\} \ y ::= V x \ \{y\} \ x ::= V z \ \{y\} \)

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

\( L \ w \ \{y\} = \{x, y\} \) is not a post-fixed point of \( L \ c: \)
\[ \{x, z\} \quad y ::= V \ x \quad \{y, z\} \quad x ::= V \ z \quad \{x, y\} \]
\( L \ c \ \{x, y\} = \{x, z\} \not\subseteq \{x, y\} \)
Define $L \omega X$ as the least post-fixed point of $\lambda P. \text{vars } b \cup X \cup L \circ P$ for true liveness
Existence of least fixed points

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

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

is the least fixed and post-fixed point of $f$. 
Proof of Knaster-Tarski

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

- \( f(\text{\textit{lfp}} f) \subseteq \text{\textit{lfp}} f \)
- \( \text{\textit{lfp}} f \) is the least post-fixed point of \( f \)
- \( \text{\textit{lfp}} f \subseteq f(\text{\textit{lfp}} f) \)
- \( \text{\textit{lfp}} f \) is the least fixed point of \( f \)
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:
$\text{lfp } f \subseteq \text{lfp } g$ if for all $X$, $f X \subseteq g X$

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

**Theorem** Let \( f :: t \text{ set} \Rightarrow t \text{ set} \). 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 post-fixed point \( p \) of \( f \) (by induction on \( i \)).
Computation of $\text{lfp } f_w$

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

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

Let \( \text{vars } c \) be the variables read in \( c \).

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

**Proof** by induction on \( c \)

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

**Corollary** \( P \subseteq V_w \implies f_w \ P \subseteq V_w \)

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

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

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

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

where \( b = \text{Less} \ (N \ 0) \ (V \ y) \)

and \( c = y ::= V \ x; \ x ::= V \ z \)

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

\[
\begin{align*}
 f_w \ \{\} & = \{y\} \cup L \ c \ \{\} = \{y\}:
 & \{\} \ y ::= V \ x \ \{\} \ x ::= V \ z \ \{\}
 & \{\} \ y ::= V \ x \ \{\} \ x ::= V \ z \ \{\}
 f_w \ \{y\} & = \{y\} \cup L \ c \ \{y\} = \{x, \ y\}:
 & \{x\} \ y ::= V \ x \ \{y\} \ x ::= V \ z \ \{y\}
 & \{x\} \ y ::= V \ x \ \{y\} \ x ::= V \ z \ \{y\}
 f_w \ \{x, \ y\} & = \{y\} \cup L \ c \ \{x, y\} = \{x, \ y, \ z\}:
 & \{x, \ z\} \ y ::= V \ x \ \{y, \ z\} \ x ::= V \ z \ \{x, \ y\}
 f_w \ \{x, \ y\} & = \{y\} \cup L \ c \ \{x, y\} = \{x, \ y, \ z\}:
 & \{x, \ z\} \ y ::= V \ x \ \{y, \ z\} \ x ::= V \ z \ \{x, \ y\}
\end{align*}
\]
An approximate approach

Fix some small $k$ (eg 3) and define

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

Is correct

**Fact** $f_w (L_w X) \subseteq L_w X$

but potentially imprecise ($V_w$).
The stabilization test $f_{w}^{i+1} \{ \} = f_{w}^{i} \{ \}$ is not directly executable in Isabelle/HOL because

- sets are functions and
- equality of functions is not executable.

Solution: implement sets by some concrete type like lists.
12 Live Variable Analysis

Soundness of $L$

Dead Variable Elimination

True Liveness

Comparisons
Comparison of analyses

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

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

Program representation:

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

- We analyse structured programs. Application: source-level program optimization.
11 Definite Assignment Analysis

12 Live Variable Analysis

13 Information Flow Analysis
The aim:

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

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

LBS guarantees confidentiality by program analysis, not by cryptography.

These analyses are often expressed as type systems.
Security levels

- Program variables have *security/confidentiality levels*.
- Security levels are partially ordered: \( l < l' \) means that \( l \) is less confidential than \( l' \).
- We identify security levels with \( \text{nat} \).
  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.
Information Flow Analysis

Secure IMP

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

Security levels:

\textbf{type\_synonym} \hspace{1em} \textit{level} = \textit{nat}

Every variable has a security level:

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

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

\textit{sec} \hspace{1em} \textit{x} = \textit{length} \hspace{1em} \textit{x}
The security level of an expression is the maximal security level of any of its variables.

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

\[
\text{sec}_\text{aexp} (N n) = 0
\]

\[
\text{sec}_\text{aexp} (V x) = \text{sec} x
\]

\[
\text{sec}_\text{aexp} (\text{Plus} \ a \ b) = \max (\text{sec}_\text{aexp} a) (\text{sec}_\text{aexp} b)
\]
Security Levels on $bexp$

$sec\_bexp :: bexp \Rightarrow level$

$sec\_bexp \ (Bc \ v) = 0$

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

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

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

Agreement of states up to a certain level:

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

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

Noninterference lemmas for expressions:

\[ s_1 = s_2 \ (\leq l) \quad sec\_aexp \ a \leq l \]
\[ aval \ a \ s_1 = aval \ a \ s_2 \]

\[ s_1 = s_2 \ (\leq l) \quad sec\_bexp \ b \leq l \]
\[ bval \ b \ s_1 = bval \ b \ s_2 \]
Information Flow Analysis

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond
Security Type System

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

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

The well-typedness predicate:

\[ l \vdash c \]

Intended meaning:

“In the context of boolean expressions of level \( \leq l \), command \( c \) is well-typed.”

Hence:

“Assignments to variables of level \( < l \) are forbidden.”
Well-typed or not?

Let \( c = \) \( \text{IF} \ \text{Less} \ (V \ "x1") \ (V \ "x") \ \text{THEN} \ "x1" ::= N \ 0 \ \text{ELSE} \ "x1" ::= N \ 1 \)

1 \( \vdash \ c \ ? \) Yes

2 \( \vdash \ c \ ? \) Yes

3 \( \vdash \ c \ ? \) No
The type system

\[ l \vdash SKIP \]
\[ sec_{-}aexp \ a \leq sec \ x \quad l \leq sec \ x \]
\[ l \vdash x ::= a \]
\[ l \vdash c_1 \quad l \vdash c_2 \]
\[ l \vdash c_1; \ c_2 \]
\[ \max (sec_{-}bexp \ b) \ l \vdash c_1 \quad \max (sec_{-}bexp \ b) \ l \vdash c_2 \]
\[ l \vdash IF \ b \ THEN \ c_1 \ ELSE \ c_2 \]
\[ \max (sec_{-}bexp \ b) \ l \vdash c \]
\[ l \vdash WHILE \ b \ DO \ c \]
Remark:

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

\[
\begin{array}{c}
l \vdash c \\
l' \leq l \\
\hline
l' \vdash c
\end{array}
\]

Proof by ... as usual.

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

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

$$
(c, s) \Rightarrow t \quad l \not| \ c
$$

$$
\frac{}{s = t \ (<_l)}
$$

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

Proof by ... 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.
Information Flow Analysis

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 $\text{max}$
- and works for arbitrary partial orders.

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

$$\begin{align*}
l \vdash' c & \quad l' \leq l \\
\hline
l' \vdash' c
\end{align*}$$

together with one rule per construct:
\[ l \vdash \text{' SKIP} \]

\[ \text{sec_aexp } a \leq \text{sec } x \quad l \leq \text{sec } x \]

\[ l \vdash \text{' } x ::= a \]

\[ l \vdash \text{' } c_1 \quad l \vdash \text{' } c_2 \]

\[ l \vdash \text{' } c_1; c_2 \]

\[ \text{sec_bexp } b \leq l \quad l \vdash \text{' } c_1 \quad l \vdash \text{' } c_2 \]

\[ l \vdash \text{' } \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \]

\[ \text{sec_bexp } b \leq l \quad l \vdash \text{' } c \]

\[ l \vdash \text{' } \text{WHILE } b \ \text{DO } c \]

400
• 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 $\vdash$. 
13 Information Flow Analysis

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

• System $\vdash c : l$ is bottom-up: $l$ is the minimal level of any variable assigned in $c$ and is checked at IF and WHILE commands.
⊢ \text{SKIP} : l

sec_aexp a \leq sec x

⊢ x ::= a : sec x

⊢ c_1 : l_1 \quad \vdash c_2 : l_2

⊢ c_1 ; c_2 : min l_1 l_2

sec_bexp b \leq min l_1 l_2 \quad \vdash c_1 : l_1 \quad \vdash c_2 : l_2

⊢ IF b THEN c_1 ELSE c_2 : min l_1 l_2

sec_bexp b \leq l \quad \vdash c : l

⊢ WHILE b DO c : l
Equivalence of $\vdash$ : and $\vdash'$

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

Proof by induction.

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

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

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

Proof by induction.
Information Flow Analysis

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?

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

Beware of covert channels!

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

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

New typing rule:

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

Now provable:

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

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

Part IV

Hoare Logic
14 Partial Correctness
15 Verification Conditions
16 Total Correctness
Partial Correctness

Verification Conditions

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:

```
"x" ::= N 0; "y" ::= N 0; w n
```

where

```
w n ≡
WHILE Less (V "y") (N n)
DO ("y" ::= Plus (V "y") (N 1);
   "x" ::= Plus (V "x") (V "y"))
```

At the end of the execution, variable "x" should contain the sum $1 + \ldots + n$. 
A proof via operational semantics

Theorem:

\((x) := N \_\_ 0; (y) := N \_\_ 0; w \_\_ n, s) \Rightarrow t \Rightarrow t x = \sum \{1..n\}\)

Required Lemma:

\((w \_\_ n, s) \Rightarrow t \Rightarrow t x = s x + \sum \{s y + 1..n\}\)

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

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

But no free lunch:

- Must prove implications between predicates on states
- Needs *invariants*. 
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 \} \textbf{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 \times (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. (Exercise!)
More rules of Hoare Logic

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

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

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

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

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

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

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

Preconditions can be strengthened, postconditions can be weakened.
Two derived rules

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

\[
P \rightarrow Q[a/x]
\]
\[
\{ P \} \quad x := a \quad \{ Q \}
\]
\[
\{ P \land b \} \quad c \quad \{ P \} \quad P \land \lnot b \rightarrow Q
\]
\[
\{ P \} \quad \text{WHILE} \quad b \quad \text{DO} \quad c \quad \{ Q \}
\]
Example

\{ True \}

\begin{align*}
  x & := 0; \ y := 0; \\
  \text{WHILE } y < n \ DO & \ (y := y+1; \ x := x+y) \\
  \{ x = \sum \{ 1..n \} \} 
\end{align*}
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.
14 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

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

\[ \iff \]

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

“\{ P \} c \{ Q \} is valid”

In contrast:

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

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

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

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

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

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

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

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

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

\[ = (\lambda s. \ s x + 5 = 5) \]
\[
\begin{align*}
\vdash \{ P \} \ c_1 \ \{ Q \} & \quad \vdash \{ Q \} \ c_2 \ \{ R \} \\
\vdash \{ P \} \ c_1; \ c_2 \ \{ R \} \\
\vdash \{ \lambda s. \ P \ s \land bval \ b \ s \} \ c_1 \ \{ Q \} \\
\vdash \{ \lambda s. \ P \ s \land \neg \ bval \ b \ s \} \ c_2 \ \{ Q \} \\
\vdash \{ P \} \ IF \ b \ THEN \ c_1 \ ELSE \ c_2 \ \{ Q \} \\
\vdash \{ \lambda s. \ P \ s \land bval \ b \ s \} \ c \ \{ P \} \\
\vdash \{ P \} \ WHILE \ b \ DO \ c \ \{ \lambda s. \ P \ s \land \neg \ bval \ b \ s \}
\end{align*}
\]
\( \forall s. \; P' \; s \rightarrow \; P \; s \)
\[ \vdash \{ P \} \; c \; \{ Q \} \]
\( \forall s. \; Q \; s \rightarrow \; Q' \; s \)
\[ \vdash \{ P' \} \; c \; \{ Q' \} \]
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: $\models \implies \vdash$
Weakest preconditions

The **weakest precondition** of command \( c \) w.r.t. postcondition \( Q \):

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

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

A foundational semantic notion, not merely for the completeness proof.
Nice and easy properties of $wp$

$wp$ $SKIP$ $Q = Q$

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

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

$wp$ ($IF \ b \ THEN \ c_1 \ ELSE \ c_2$) $Q =$

$(\lambda s. \ (bval \ b \ s \rightarrow \ wp \ c_1 \ Q \ s) \land$

$(\neg \ bval \ b \ s \rightarrow \ wp \ c_2 \ Q \ s))$

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

$bval \ b \ s \rightarrow$

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

$wp \ (c; \ WHILE \ b \ DO \ c) \ Q \ s$
Completeness

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

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

**Lemma** \( \vdash \{ \text{wp} \ c \ Q \} \ c \ \{ Q \} \)

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

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

**Fact** \( \models \{ P \} \ c \ \{ Q \} \implies \forall s. P \ s \implies \text{wp} \ c \ Q \ s \)

Follows directly from defs of \( \models \) and \( \text{wp} \).
Proving program properties by Hoare logic (⊢) is just as powerful as by operational semantics (|=).
WARNING

Most texts that discuss completeness of Hoare logic state or prove that Hoare logic is only “relatively complete” but not complete.
Reason: the standard notion of completeness assumes some abstract mathematical notion of $\models$.
Our notion of $\models$ is defined within the same (limited) proof system (for HOL) as $\vdash$. 
14 Partial Correctness

15 Verification Conditions

16 Total Correctness
Idea:

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

More precisely:

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

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

Method:

Simulate syntax-directed application of Hoare logic rules. Collect all assertion language side conditions.
A problem: loop invariants

Where do they come from?

A trivial solution:

Let the user provide them!

How?

Each loop must be annotated with its invariant!
How to synthesize loop invariants automatically is an important research problem. Which we ignore for the moment. But come back to later.
Terminology:

\[ \text{VCG} = \text{Verification Condition Generator} \]

All successful verification technology for imperative programs relies on

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

1. Introduce annotated commands with loop invariants

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

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

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

The details are a bit different . . .
Annotated commands

Like commands, except for *While*:

**datatype** \( a \text{com} \) = \( \text{ASKIP} \)

\| \( \text{Assign } \text{vname } \text{aexp} \)
\| \( \text{Asemi } \text{acom } \text{acom} \)
\| \( \text{Aif } \text{bexp } \text{acom } \text{acom} \)
\| \( \text{Awhile } \text{assn } \text{bexp } \text{acom} \)

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

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

\( \text{pre :: acom } \Rightarrow \text{ assn } \Rightarrow \text{ assn} \)

\( \text{pre ASKIP } Q = Q \)

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

\( \text{pre } (c_1; \ c_2) \ Q = \text{pre } c_1 (\text{pre } c_2 \ Q) \)

\( \text{pre } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q = (\lambda s. \ (bval b \ s \rightarrow \text{pre } c_1 \ Q \ s) \land \neg bval b \ s \rightarrow \text{pre } c_2 \ Q \ s)) \)

\( \text{pre } ('\{I\} \ \text{WHILE } b \ \text{DO } c) \ Q = I \)
Warning

In the presence of loops, 

\(\text{pre} \ c\) may not be the weakest precondition but may be anything!
Verification condition

\( vc :: a\text{com} \Rightarrow a\text{ssn} \Rightarrow a\text{ssn} \)

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

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

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

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

\( vc \ ({\{I\}} \ WHILE \ b \ DO \ c) \ Q = \\
(\lambda s. \ (I \ s \land \neg bval \ b \ s \rightarrow Q \ s) \land \\
(I \ s \land bval \ b \ s \rightarrow pre \ c \ I \ s) \land \ vc \ c \ I \ s) \)
Verification conditions only arise from loops:
- the invariant must be invariant
- and it must imply the postcondition.

Everything else in the definition of $v_c$ is just bureaucracy: collecting assertions and passing them around.
Hoare triples operate on $com$, functions $pre$ and $vc$ operate on $acom$. Therefore we define

$$strip :: acom \Rightarrow com$$

$$strip \ ASKIP = SKIP$$

$$strip \ (x ::= a) = x ::= a$$

$$strip \ (c_1; c_2) = strip \ c_1; \ strip \ c_2$$

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

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

\[ \forall s. \, vc \, c \, Q \, s \implies \vdash \{pre \, c \, Q\} \, strip \, c \, \{Q\} \]

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

Corollary:

\[ (\forall s. \, vc \, c \, Q \, s) \land (\forall s. \, P \, s \implies pre \, c \, Q \, s) \implies \vdash \{P\} \, strip \, c \, \{Q\} \]

How to prove some \( \vdash \{P\} \, c_0 \, \{Q\} \):

- Annotate \( c_0 \) yielding \( c \), i.e. \( \text{strip} \, c = c_0 \).
- Prove Hoare-free premise of corollary.

But is premise provable if \( \vdash \{P\} \, c_0 \, \{Q\} \) is?
\[(\forall s. \ vc \ c \ Q \ s) \land (\forall s. \ P \ s \to \ pre \ c \ Q \ s) \Rightarrow \]
\[\vdash \{P\} \ strip \ c \ \{Q\}\]

Why could premise not be provable although conclusion is?

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

Therefore we prove completeness: suitable annotations exist such that premise is provable.
Completeness of $\textit{vc} \& \textit{pre}$ w.r.t. $\vdash$

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

\[\exists \ c'. \ \textit{strip} \ c' = c \ \land \]

\[(\forall s. \ \textit{vc} \ c' \ Q \ s) \ \land \ (\forall s. \ P \ s \implies \textit{pre} \ c' \ Q \ s)\]

Proof by rule induction. Needs two monotonicity lemmas:

\[[\forall s. \ P \ s \implies P' \ s; \ \textit{pre} \ c \ P \ s] \implies \textit{pre} \ c \ P' \ s\]

\[[\forall s. \ P \ s \implies P' \ s; \ \textit{vc} \ c \ P \ s] \implies \textit{vc} \ c \ P' \ s\]
14 Partial Correctness

15 Verification Conditions

16 Total Correctness
• **Partial Correctness:**
  if command terminates, postcondition holds

• **Total Correctness:**
  command terminates *and* postcondition holds

**Total Correctness = Partial Correctness + Termination**

Formally:

\[
\models_t \{ P \} \ c \ \{ Q \} \equiv \forall s. \ P_s \rightarrow (∃ t. (c, s) \Rightarrow t \land Q_t)
\]

Assumes that semantics is deterministic!

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

Only need to change the While-rule.

Some measure function state ⇒ nat must decrease with every loop iteration

\[ \bigwedge n. \vdashₜ \{ \lambda s. \ P \ s \land bval \ b \ s \land f \ s = n \} \ c \ \{ \lambda s. \ P \ s \land f \ s < n \} \]

\[ \vdashₜ \{ P \} \ \text{WHILE} \ b \ \text{DO} \ c \ \{ \lambda s. \ P \ s \land \neg \ bval \ b \ s \} \]
HoareT.thy

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

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

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

Follows easily from

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

where

\[ \text{wp}_t \ c \ Q \equiv \lambda s. \ \exists t. \ (c, \ s) \Rightarrow t \land Q \ t. \]
Proof of $\vdash_t \{wp_t \ c \ Q\} \ c \ \{Q\}$ is by induction on $c$.

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

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

Abstract Interpretation
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Computable Abstract State

Backward Analysis of Boolean Expressions

Widening and Narrowing
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Computable Abstract State

Backward Analysis of Boolean Expressions

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

- Constant folding
- Unreachable and dead code elimination
- Array access optimization:
  
  \[
  a[i] := 1; \quad a[j] := 2; \quad x := a[i] \\
  a[i] := 1; \quad a[j] := 2; \quad x := 1 \quad \text{if} \quad i \neq j
  \]

- ...

\[
\]
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/error/exception
• cannot arise, this is definitely the case.
• can arise, this is only potentially the case.
Beware of false alarms because of overapproximation.
Error

Program Analysis

No Alarm

False Alarm

True Alarm
The starting point: Collecting Semantics

Collects all possible states for each program point:

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

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

\[
\begin{align*}
x & := 0; \ y := 0 \ \{ \ <x:=0, \ y:=0> \ \} ; \\
\{ \ <x:=0, \ y:=0>, \ <x:=2, \ y:=1>, \ <x:=4, \ y:=2> \ \} \\
\text{WHILE} \ x < 3 \ \text{DO} \\
\quad x := x+2; \ y := y+1 \\
\quad \{ \ <x:=2, \ y:=1>, \ <x:=4, \ y:=2> \ \} \\
\{ \ <x:=4, \ y:=2> \ \}
\end{align*}
\]
A first approximation

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

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

WHILE \(x < 3\) DO

\[
x := x + 2 \{ <x := \{2, 4\}> \}
\{ <x := \{4\}> \}
\]
Loses relationships between variables but simplifies matters a lot.

Example:

\[
\{ \langle x := 0, \ y := 0 \rangle, \ \langle x := 1, \ y := 1 \rangle \} 
\]

is approximated by

\[
\langle x := \{0, 1\}, \ y := \{0, 1\} \rangle 
\]

which also subsumes

\[
\langle x := 0, \ y := 1 \rangle \ \text{and} \ \langle x := 1, \ y := 0 \rangle.
\]
Abstract Interpretation

Approximate sets of concrete values by *abstract values*

Example: approximate sets of numbers by intervals

Execute/interpret program with abstract values
A consistently annotated program:

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

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
Control Flow Graph (CFG)

View command as graph where edges are labeled with atomic commands (SKIP, \( x:=a \)) or conditions:

\[
\begin{align*}
&x:=0 \\
\neg x<3 &\quad x<3 \\
\end{align*}
\]

\[
\begin{align*}
\hspace{1cm} &\quad \hspace{1cm} x:=x+2
\end{align*}
\]

In an annotated command/CFG, the nodes are labeled, for example with sets of states.
Annotated commands

Concrete syntax:

\[
'a\ acom ::= \\
\quad \text{SKIP} \quad \{ \ 'a \ \} \\
\mid \text{string} ::= \text{aexp} \quad \{ \ 'a \ \} \\
\mid 'a\ acom ; 'a\ acom \\
\mid \text{IF bexp THEN 'a acom ELSE 'a acom} \quad \{ \ 'a \ \} \\
\mid \{ \ 'a \ \} \ \text{WHILE bexp DO 'a acom} \quad \{ \ 'a \ \}
\]

'a: type of annotations

Example: "x" ::= N 1 \{ 9 \}; SKIP \{ 6 \} :: nat acom
Annotated commands

Abstract syntax:

```
datatype 'a acom =
    SKIP 'a
  | Assign string aexp 'a
  | Semi ('a acom) ('a acom)
  | If bexp ('a acom) ('a acom) 'a
  | While 'a bexp ('a acom) 'a
```
Auxiliary functions: $post$

\[
\begin{align*}
\text{post} &::= \ 'a \ acom \Rightarrow \ 'a \\
\text{post} \ (\text{SKIP} \ \{P\}) & = P \\
\text{post} \ (x ::= e \ \{P\}) & = P \\
\text{post} \ (c_1; \ c_2) & = \text{post} \ c_2 \\
\text{post} \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \{P\}) & = P \\
\text{post} \ (\{\text{Inv}\} \ \text{WHILE} \ b \ \text{DO} \ c \ \{P\}) & = P
\end{align*}
\]
Auxiliary functions: \textit{strip}

\begin{align*}
\textit{strip} :: 'a \text{ acom} & \Rightarrow \text{ com} \\
\textit{strip} (\text{SKIP} \{P\}) & = \text{SKIP} \\
\textit{strip} (x ::= e \{P\}) & = x ::= e \\
\textit{strip} (c_1; c_2) & = \textit{strip} c_1; \textit{strip} c_2 \\
\textit{strip} (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \{P\}) & = \text{IF} \ b \ \text{THEN} \ \textit{strip} c_1 \ \text{ELSE} \ \textit{strip} c_2 \\
\textit{strip} (\{\text{Inv}\} \ \text{WHILE} \ b \ \text{DO} \ c \ \{P\}) & = \text{WHILE} \ b \ \text{DO} \ \textit{strip} c
\end{align*}

We call \(c\) and \(c'\) \textit{strip-equal} iff \(\textit{strip} c = \textit{strip} c'\).
Auxiliary functions: \textit{anno}

\[
anno :: \ 'a \Rightarrow \text{com} \Rightarrow \ 'a \ acom
\]

\[
anno a \ SKIP = SKIP \ \{ a \}
\]

\[
anno a \ (x ::= e) = x ::= e \ \{ a \}
\]

\[
anno a \ (c_1; c_2) = anno a \ c_1; \ anno a \ c_2
\]

\[
anno a \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2)
\quad = \ IF \ b \ \text{THEN} \ anno a \ c_1 \ \text{ELSE} \ anno a \ c_2 \ \{ a \}
\]

\[
anno a \ (\text{WHILE} \ b \ \text{DO} \ c)
\quad = \ \{ a \} \ \text{WHILE} \ b \ \text{DO} \ anno a \ c \ \{ a \}
\]
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Computable Abstract State

Backward Analysis of Boolean Expressions

Widening and Narrowing
Annotate commands with the set of states that can occur at each annotation point, i.e. behind each command and in front of loops.

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.

Start states flowing into the command
\text{step}

\text{step } S \ (\text{SKIP \ \{} \_ \}) = \text{SKIP \ \{} S \} \\
\text{step } S \ (x ::= e \ \{} \_ \) = \\
x ::= e \ \{} \{ \{ s' \colon \exists s \in S. \ s' = s(x ::= \text{aval } e \ s) \} \} \\
\text{step } S \ (c_1; c_2) = \text{step } S \ c_1; \text{step } (\text{post } c_1) \ c_2 \\
\text{step } S \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \{} \_ \) = \\
\text{IF } b \ \text{THEN } \text{step } \{ s \in S. \ \text{bval } b \ s \} \ c_1 \\
\text{ELSE } \text{step } \{ s \in S. \ \neg \ \text{bval } b \ s \} \ c_2 \\
\{ \text{post } c_1 \cup \text{post } c_2 \}
$$\text{step } S \left( \{ \text{Inv} \} \ \text{WHILE } b \ \text{DO } c \ \{ \_ \} \right) = \{ S \cup \text{post } c \}$$

$$\text{WHILE } b \ \text{DO } \text{step } \{ s \in \text{Inv. } \text{bval } b \ s \} \ c$$

$$\{ \{ s \in \text{Inv. } \neg \text{bval } b \ s \} \}$$
Collecting semantics

View command as CFG

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

Stabilization means fixed point:

$$\text{step } S \ c = c$$
Collecting_list.thy

Examples
Abstract example

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

WHILE \( x < 3 \) DO
  \( x := x+2 \) \( \{ A \} \)
  \( \{ P \} \)

**step** \( S \ c = c \) means

\[
I = S \cup A \\
A = \{ s'. \exists s \in I. \ bval b s \land s' = s(x := s x + 2) \} \\
P = \{ s \in I. \neg bval b s \}
\]

**Fixed point = solution of equation system**

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

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

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

But the “reachable” fixed point is \( I = \{ \} \)
Complete lattice

Definition
A type 'a with a partial order ≤ 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 $\bigcap$ is the infimum.
**Lemma** In a complete lattice, every set $S$ of elements also has a least upper bound (supremum) $\bigsqcup S$:

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

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

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

**Definition** A function $f$ on a partial order $\leq$ is monotone if $x \leq y \implies f(x) \leq f(y)$.

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

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

**Proof** just like the version for sets.
Any ordering on 'a can be lifted to 'a acom by comparing the annotations of strip-equal commands:

\[\text{SKIP}\{S\} \leq \text{SKIP}\{S'\} \iff S \leq S'\]

\[x ::= e \{S\} \leq x' ::= e' \{S'\} \iff x = x' \land e = e' \land S \leq S'\]

\[c_1; c_2 \leq d_1; d_2 \iff c_1 \leq d_1 \land c_2 \leq d_2\]

\[\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \{S\} \leq \text{IF } b' \text{ THEN } d_1 \text{ ELSE } d_2 \{S'\} \iff b = b' \land c_1 \leq d_1 \land c_2 \leq d_2 \land S \leq S'\]

\[\{I\} \text{ WHILE } b \text{ DO } c \{P\} \leq \{I'\} \text{ WHILE } b' \text{ DO } c' \{P'\} \iff b = b' \land c \leq c' \land I \leq I' \land P \leq P'\]
For all other (not strip-equal) commands:

\[ c \leq c' \iff False \]

Example:

\[
\begin{align*}
  x &::= N 0 \{\{a\}\} \leq x ::= N 0 \{\{a, b\}\} \iff True \\
  x &::= N 0 \{\{a\}\} \leq x ::= N 0 \{\{\}\} \iff False \\
  x &::= N 0 \{S\} \leq x ::= N 1 \{S\} \iff False
\end{align*}
\]
The collecting semantics needs to order \textit{state set acom}.

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

Does \textit{state set acom} also form a complete lattice? Almost . . .
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 $\text{strip}$-equal commands have an infimum
It turns out:

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

Therefore we index our complete lattices.
Indexed Complete Lattice

Definition A partially ordered type \( 'a \) is a complete lattice indexed by type \( 'i \)

- if there is a function \( L :: 'i \Rightarrow 'a \text{ set} \) such that
- for every \( i :: 'i \) and \( M \subseteq L \ i \)
- \( M \) has a greatest lower bound \( \bigcap_i M \in L \ i \).
Application to \textit{acom}

How to view \textit{'}a acom} (where \textit{'}a} is a complete lattice) as a complete lattice indexed by \textit{com}:

- \( L(c :: \text{com}) = \{c' :: \text{'}a acom. \text{strip } c' = c\} \)
- The infimum of a set \( M \subseteq L c \) is computed “pointwise”:

  Annotate \( c \) at program point \( p \) with the infimum of the annotations of all \( c' \in M \) at \( p \).

Example \( \bigsqcap \text{SKIP} \{\text{SKIP } \{A\}, \text{SKIP } \{B\}, \ldots \} \)

\[ = \text{SKIP } \{\bigsqcap \{A,B, \ldots \}\} \]

Formally \ldots
Some auxiliary functions:

The image of a set $A$ under a function $f$:

$$f \upharpoonright A = \{y. \exists x \in A. y = f(x)\}$$

Predefined in HOL.

Selecting subcommands:

$$\text{sub}_1 \ (c_1; c_2) = c_1$$
$$\text{sub}_1 \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \{S\}) = c_1$$
$$\text{sub}_1 \ (\{I\} \ \text{WHILE } b \ \text{DO } c \ \{P\}) = c$$

$$\text{sub}_2 \ (c_1; c_2) = c_2$$
$$\text{sub}_2 \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \{S\}) = c_2$$

Selecting the invariant:

$$\text{invar} \ (\{I\} \ \text{WHILE } b \ \text{DO } c \ \{P\}) = I$$
How to lift some $F :: 'a set \Rightarrow 'a$:

\[
\begin{align*}
\text{lift} & :: ( 'a set \Rightarrow 'a ) \Rightarrow \text{com} \Rightarrow 'a \ acom \ set \Rightarrow 'a \ acom \\
\text{lift } F \ SKIP \ M & = \ SKIP \ \{ F ( post ' M ) \} \\
\text{lift } F \ (x ::= a) \ M & = \ x ::= a \ \{ F ( post ' M ) \} \\
\text{lift } F \ (c_1 ; c_2) \ M & = \\
& \quad \text{lift } F \ c_1 \ (sub_1 ' M) ; \text{lift } F \ c_2 \ (sub_2 ' M) \\
\text{lift } F \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ M & = \\
& IF \ b \ THEN \ \text{lift } F \ c_1 \ (sub_1 ' M) \\
& ELSE \ \text{lift } F \ c_2 \ (sub_2 ' M) \\
& \{ F ( post ' M ) \} \\
\text{lift } F \ (WHILE \ b \ DO \ c) \ M & = \\
& \{ F ( invar ' M ) \} \\
& WHILE \ b \ DO \ \text{lift } F \ c \ (sub_1 ' M) \\
& \{ F ( post ' M ) \}
\end{align*}
\]
Lemma If \( 'a \) is a complete lattice, then \( 'a acom \) is a complete lattice indexed by \( \text{com} \) where the infimum of \( M \subseteq L c \) is

\[
\bigcap_c M = \text{lift} \bigcap_c c M
\]

Proof of the infimum properties of \( \bigcap_c M \) by induction on \( c \).
We say that $f$ preserves $L$ if $\forall i. f \ ' L_i \subseteq L_i$.

**Theorem** Let $'a$ be a complete lattice indexed by $'i$. If $f :: 'a \Rightarrow 'a$ is monotone and preserves $L$, then for every $i :: 'i$, $f$ (restricted to $L_i$) has the least (post-)fixed point

$$lfp\ f\ i = \bigcap_i \{p \in L_i. f\ p \leq p\}.$$

**Proof** just like for the standard version.
The Collecting Semantics

**Lemma** \(\text{step } S\) is monotone and preserves \(L\).

Therefore Knaster-Tarski is applicable and we define

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

\[
CS \ c = \text{lfp } (\text{step UNIV}) \ c
\]
17 Introduction
18 Annotated Commands
19 Collecting Semantics
20 Abstract Interpretation: Orderings
21 A Generic Abstract Interpreter
22 Computable Abstract State
23 Backward Analysis of Boolean Expressions
24 Widening and Narrowing
Approximating the Collecting semantics

A conceptual step:

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

A domain-specific step:

\[\text{val set} \quad \sim \quad 'av\]

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

Abstract values:

```
datatype parity = Even | Odd | Either
```

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

Bigger abstract values represent more concrete values.
A type 'a is a preorder if

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

A partial order is also antisymmetric
\( ([x \sqsubseteq y; y \sqsubseteq x] \Rightarrow x = y) \)
Partial orders are technically simpler.

Preorders are more liberal:

- they allow different representations for the same abstract element.
  
  **Example:** the intervals $[1,0]$ and $[2,0]$ both represent the empty interval.

- Instead of $x = y$, test for $x \subseteq y \land y \subseteq x$. 
Example: parity

Fact Type \textit{parity} is a partial order.
A type `a is a semilattice with top element if

- it is a preorder and
- there is a least upper bound operation
  \[ \sqcup :: 'a \Rightarrow 'a \Rightarrow 'a \]
  \[ x \sqsubseteq x \sqcup y \quad y \sqsubseteq x \sqcup y \]
  \[ [x \sqsubseteq z; y \sqsubseteq z] \Longrightarrow x \sqcup y \sqsubseteq z \]

- and a top element \( \top :: 'a \)
  \[ x \sqsubseteq \top \]

Application: abstract \( \sqcup \), join two computation paths
We often call \( \sqcup \) the `join` operation.
**Lemma** If \( a \) is a semilattice where \( \sqsubseteq \) is actually a partial order, then the least upper bound of two elements is uniquely determined (and similarly the top element).

\( \sqsubseteq \) uniquely determines \( \sqcup \) and \( \top \)
Example: parity

\[
\begin{align*}
\text{Either} & \\
\quad / & \\
\quad / & \\
\text{Even} & \quad \text{Odd}
\end{align*}
\]

Fact Type \textit{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 \texttt{preord}: preorders
- class \texttt{SL\_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: \( \text{parity} :: \text{SL\_top} \)
Abs_Int0_fun
Abs_Int0_parity.thy

Orderings
From abstract values to abstract states

Need to abstract collecting semantics:

\[ \text{state set} \]

- First attempt:

\[ 'av \ st = \ vname \Rightarrow 'av \]

where \( 'av \) is the type of abstract values

- Problem: cannot abstract empty set of states (unreachable program points!)

- Solution: type \( 'av \ st \ option \)
Concretization functions

Let $\gamma :: \text{'av} \Rightarrow \text{val set}$

Define

$\gamma_f :: \text{'av st} \Rightarrow \text{state set}$

$\gamma_f S = \{ s. \forall x. s \ x \in \gamma(S \ x) \}$

$\gamma_o :: \text{'av st option} \Rightarrow \text{state set}$

$\gamma_o \ None = \{\}$

$\gamma_o \ (\text{Some} \ S) = \gamma_f S$
Lemma If 'a :: SL_top then 'b ⇒ 'a :: SL_top.
Proof
(f ⊑ g) = (∀ x. f x ⊑ g x)
f ⊔ g = (λx. f x ⊔ g x)
⊤ = (λx. ⊤)
'av st option as a semilattice

Lemma If 'a :: SL_top then 'a option :: SL_top.

Proof

\((\text{Some } x \sqsubseteq \text{Some } y) = (x \sqsubseteq y)\)
\((\text{None } \sqsubseteq \_ ) = \text{True}\)
\((\text{Some } \_ \sqsubseteq \text{None} ) = \text{False}\)

\(\text{Some } x \sqcup \text{Some } y = \text{Some } (x \sqcup y)\)
\(\text{None } \sqcup y = y\)
\(x \sqcup \text{None} = x\)
\(\top = \text{Some } \top\)

Corollary If 'a :: SL_top then 'a st option :: SL_top.
Lemma If \( 'a :: \text{preord} \) then \( 'a \text{acom} :: \text{preord} \).

Proof \( \sqsubseteq \) is lifted from \( 'a \) to \( 'a \text{acom} \) just like \( \leq \).
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Computable Abstract State

Backward Analysis of Boolean Expressions

Widening and Narrowing
• Stepwise development of a generic abstract interpreter as a parameterized module
• Parameters/Input: abstract type of values together with abstractions of the operations on concrete type \( \text{val} = \text{int} \).
• Result/Output: abstract interpreter that approximates the collecting semantics by computing on abstract values.
• Realization in Isabelle as a locale
Parameters (I)

Abstract values: type \( 'av :: SL_{top} \)
Concretization function: \( \gamma :: 'av \Rightarrow val\ set \)
Assumptions: \( a \subseteq b \implies \gamma a \subseteq \gamma b \)
\( \gamma \top = UNIV \)
Parameters (II)

Abstract arithmetic:  \( \text{num}' :: \text{val} \Rightarrow \text{'av} \)
\( \text{plus}' :: \text{'av} \Rightarrow \text{'av} \Rightarrow \text{'av} \)

Intention:  \( \text{num}' \) abstracts the meaning of  \( N \)
\( \text{plus}' \) abstracts the meaning of  \( \text{Plus} \)

Required for each constructor of  \( \text{aexp} \) (except  \( V \))

Assumptions:
\( n \in \gamma (\text{num}' n) \)
\[ [n_1 \in \gamma a_1; n_2 \in \gamma a_2] \implies n_1 + n_2 \in \gamma (\text{plus}' a_1 a_2) \]

The  \( n \in \gamma a \) relationship is maintained
Abstract interpretation of \( aexp \)

\[
\text{fun } \text{aval}' :: aexp \Rightarrow 'av \text{ st } \Rightarrow 'av
\]

\[
\text{aval}' (N \ n) \ S = \text{num}' \ n
\]

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

\[
\text{aval}' (\text{Plus} \ a_1 \ a_2) \ S = \text{plus}' (\text{aval}' a_1 \ S) (\text{aval}' a_2 \ S)
\]

Correctness of \( \text{aval}' \) wrt \( \text{aval} \):

**Lemma** \( s \in \gamma_f \ S \iff \text{aval} a s \in \gamma (\text{aval}' a S) \)

**Proof** by induction on \( a \) using the assumptions about the parameters.
Example instantiation with \textit{parity}

\( \mathbb{E}/\mathbb{O} \) and \( \gamma_{\text{parity}} \): see earlier

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

\textit{plus\_parity} \( \text{Even Even} = \text{Even} \)
\( \text{Odd Odd} = \text{Even} \)
\( \text{Even Odd} = \text{Odd} \)
\( \text{Odd Even} = \text{Odd} \)
\( \text{Either y} = \text{Either} \)
\( x \text{ Either} = \text{Either} \)
Example instantiation with \textit{parity}

Input: \( \gamma \mapsto \gamma_{\text{parity}} \)

\( \text{num}' \mapsto \text{num}_{\text{parity}} \)

\( \text{plus}' \mapsto \text{plus}_{\text{parity}} \)

Must prove parameter assumptions

Output: \( \text{aval}' \mapsto \text{aval}_{\text{parity}} \)

Example The value of

\[
\text{aval}_{\text{parity}} (\text{Plus (V "x") (V "x"))}
\]

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

is \textit{Even}. 
Abs_Int0_parity.thy

Locale interpretation
Abstract interpretation of $bexp$

For now, boolean expressions are not analysed.
Abstract interpretation of $com$

Abstracting the collecting semantics

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

$step' :: 'av\ st\ option \Rightarrow$

$'av\ st\ option\ acom \Rightarrow 'av\ st\ option\ acom$
\[ \text{step'} \ S \ (\text{SKIP} \ \{\_\}) = \text{SKIP} \ \{S\} \]

\[ \text{step'} \ S \ (x ::= \ e \ \{\_\}) = \]
\[ x ::= \ e \]
\[ \{\text{case} \ S \ \text{of} \ \\ None \Rightarrow \ None \ \\ Some \ S \Rightarrow Some \ (S(x ::= \ \text{aval'} \ e \ S))\} \]

\[ \text{step'} \ S \ (c_1; \ c_2) = \text{step'} \ S \ c_1; \ \text{step'} \ (\text{post} \ c_1) \ c_2 \]

\[ \text{step'} \ S \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \{\_\}) = \]
\[ \text{IF} \ b \ \text{THEN} \ \text{step'} \ S \ c_1 \ \text{ELSE} \ \text{step'} \ S \ c_2 \ \\
\{\text{post} \ c_1 \sqcap \text{post} \ c_2\} \]

\[ \text{step'} \ S \ (\{\text{Inv}\} \ \text{WHILE} \ b \ \text{DO} \ c \ \{\_\}) = \]
\[ \{S \sqcap \text{post} \ c\} \ \text{WHILE} \ b \ \text{DO} \ \text{step'} \ \text{Inv} \ c \ \{\text{Inv}\} \]
Example: iterating \textit{step\_parity}

\[(\textit{step\_parity } S)^k \; c\]

where

\[c = \begin{align*}
&x ::= N \; 3 \; \{\text{None}\} \; ; \\
&\{\text{None}\} \\
\textbf{WHILE} \; b \; \textbf{DO} \\
&x ::= \text{Plus} \; (V \; x) \; (N \; 5) \; \{\text{None}\} \\
&\{\text{None}\}
\end{align*}\]

\[S = \text{Some} \; (\lambda_. \; \text{Either})\]

\[S_p = \text{Some} \; ((\lambda_. \; \text{Either})(x := p))\]
Correctness of $step'$ wrt $step$

$step$ and $step'$ proceed in lock-step:
If the arguments are related, so are the results.

**Lemma** If $S \subseteq \gamma_o S'$ and $c \leq \gamma_c c'$
then $step S c \leq \gamma_c (step' S' c')$

where $S :: state\ set$, $S' :: \ 'av\ st\ option$
$c :: state\ set\ acom$, $c' :: \ 'av\ st\ option\ acom$

$\gamma_c :: \ 'av\ st\ option\ acom \Rightarrow \ state\ set\ acom$
$\gamma_c = map\_acom \gamma_o$

**Proof** by induction on $c$ (or $c'$)
The abstract interpreter

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

From the HOL library:

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

such that

\[ \text{while\_option } b \ c \ s = \]
\[ (\text{if } b \ s \text{ then } \text{while\_option } b \ c \ (c \ s) \text{ else } \text{Some } s) \]

and \[ \text{while\_option } b \ c \ s = \text{None} \]

if the recursion does not terminate.
Post-fixed point:

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

\[ pfp \ f = \text{while\_option} (\lambda x. \neg f \ x \sqsubseteq x) \ f \]

Least post-fixed point on annotated commands:

\[ lpfp_c :: (\texttt{a} \texttt{ option acom} \Rightarrow \texttt{a} \texttt{ option acom}) \]

\[ \Rightarrow \texttt{com} \Rightarrow \texttt{a} \texttt{ option acom option} \]

\[ lpfp_c \ f \ c = pfp \ f (\bot_c \ c) \text{ where } \bot_c = \text{anno None} \]

N.B. \( \bot_c \ c \) is least \( \texttt{a} \texttt{ option acom} \) wrt \( \sqsubseteq \)
The generic abstract interpreter

definition $AI :: \text{com} \Rightarrow 'av \text{st option acom option}$
where $AI = \text{lpfp}_c (\text{step'} \top)$

Theorem $AI c = \text{Some } c' \iff CS c \leq \gamma_c c'$

Proof From the assumption: $\text{step'} \top c' \subseteq c'$.
Because $CS$ is a least (post-)fixed point: show that $\gamma_c (\text{step'} \top c')$ is a post-fixed point of $\text{step UNIV}$, using the correctness of $\text{step'}$ wrt $\text{step}$ and $\gamma_c (\text{step'} \top c') \leq \gamma_c c'$ (monotonicity of all $\gamma$s)
Problem

*AI* is not directly executable

because \( pfp \) compares \( f \; c \sqsubseteq c \)

where \( c :: \text{'av st option acom} \)

which compares functions \( vname \Rightarrow \text{'av} \)

which is (in general) uncomputable: \( vname \) is infinite.
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Computable Abstract State

Backward Analysis of Boolean Expressions

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

Thus we replace \( 'av \ st = \ vname \Rightarrow 'av \) by

**datatype** \( 'av \ st = \)

\[
\text{FunDom} \ (vname \Rightarrow 'av) \ (vname \ list)
\]

where \( \text{FunDom} \ f \ xs \) represents a function \( f \) with an explicit domain \( xs \) (which is necessarily finite).

Many other (more efficient) representations are possible.
Projections: \[ \text{fun } (\text{FunDom } f \ _ ) = f \]
\[ \text{dom } (\text{FunDom } \ _ xs) = xs \]

Explicit function application:
\[ \text{lookup } F \ x = (\text{if } x \in \text{set } (\text{dom } F) \ \text{then } \text{fun } F \ x \ \text{else } \top) \]

Variables outside \( \text{dom} \) are mapped to \( \top \)

\[ \text{update } F \ x \ y = \]
\[ \text{FunDom } ((\text{fun } F)(x := y)) \]
\[ (\text{if } x \in \text{set } (\text{dom } F) \ \text{then } \text{dom } F \ \text{else } x \neq \text{dom } F) \]

Concretization:
\[ \gamma_f F = \{ f. \ \forall x. \ f \ x \in \gamma (\text{lookup } F \ x) \}\]
'av st as a semilattice

Lemma If 'a :: SL_top then 'a st :: SL_top.

Proof
(F ⊑ G) = (∀ x∈set (dom G). lookup F x ⊑ fun G x)

F ⊔ G = FunDom (λx. fun F x ⊔ fun G x)
   (inter_list (dom F) (dom G))

⊤ = FunDom (λx. ⊤) []
The generic abstract interpreter

Everything as before, except

- new definition of 'av st
- \( S \ x \sim \text{lookup } S \ x \)
- \( S(x := a) \sim \text{update } S \ x \ a \)

Now \( \sqsubseteq \) on 'av st is computable.
Abs_Int0_parity.thy

Examples
Abs_Int0_const.thy
Monotonicity

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

\[
[a_1 \sqsubseteq b_1; a_2 \sqsubseteq b_2] \implies \text{plus'} a_1 a_2 \sqsubseteq \text{plus'} b_1 b_2
\]

**Theorem** In the monotone framework, \( \text{aval'} \) is also monotone

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

and therefore \( \text{step'} \) is also monotone:

\[
[S_1 \sqsubseteq S_2; c_1 \sqsubseteq c_2] \implies \text{step'} S_1 c_1 \sqsubseteq \text{step'} S_2 c_2
\]
Termination

Definition \( x \sqsubseteq y \iff x \sqsubseteq y \land \neg y \sqsubseteq x \)

Definition \( \sqsubseteq \) satisfies the ascending chain condition iff there is no infinite ascending chain \( x_0 \sqsubseteq x_1 \sqsubseteq \ldots \)

Theorem In the monotone framework:
If \( \sqsubseteq \) on \( 'av \) satisfies the ascending chain condition then \( AI \) terminates: \( \exists c'. AI c = Some c' \).
Proof sketch: Because \( \text{step}' \) is monotone, starting from \( \bot_c c \) generates an ascending \( \sqsubset \) chain of annotated commands. Each \( \sqsubset \) step on \( a\text{com} \) means \( \sqsubseteq \) for all annotations and \( \sqsubset \) for at least one annotation. This annotation either changes from \( \text{None} \) to \( \text{Some} \) (this can only happen finitely often), or from \( \text{Some } S \) to \( \text{Some } S' \) such that there is one \( x \) such that \( \text{lookup } S \ x \sqsubset \text{lookup } S' \ x \). Hence an infinite ascending chain on \( a\text{com} \) would induce and infinite ascending chain on \( \text{'av} \), a contradiction.
A simple proof of the ascending chain condition: find measure function \( m :: \text{'av} \Rightarrow \text{nat} \) such that

- \( x \sqsubseteq y \implies m x > m y \)
- \( x \sqsubseteq y \land y \sqsubseteq x \implies m x = m y \)

In practice we want something even stronger: \( \sqsubseteq \) is of finite height: \( m x < h \) (parity: \( h = 2 \))

Then \( \text{AI } c \) needs at most \( O(p n h) \) steps where
- \( p = \) number of annotations in \( c \)
- \( n = \) number of variables in \( c \)

Note: wellfoundedness means no infinite descending chains
Warning: \textit{step}' is very inefficient.
It is applied to every subcommand in every step.

Better iteration policy:
Ignore subcommands where nothing has changed.

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

As usual: \textit{efficiency complicates proofs.}
Introduction

Annotated Commands

Collecting Semantics

Abstract Interpretation: Orderings

A Generic Abstract Interpreter

Computable Abstract State

Backward Analysis of Boolean Expressions

Widening and Narrowing
Need to simulate collecting semantics ($S :: state set$):

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

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

\[
\text{if } s \in \gamma_f S \text{ and } bval b s \text{ then } s \in \gamma_f S'\]

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

Computing $S'$ from $S$ requires $\sqcap$
A type \('a\) is a **lattice with top and bottom** if

- it is a semilattice with top
- there is a greatest lower bound operation
  \(\sqcap: \quad \forall \ a \Rightarrow \ a \Rightarrow \ a\)
  \(x \sqcap y \sqsubseteq x \quad x \sqcap y \sqsubseteq y\)
  \([z \sqsubseteq x; \ z \sqsubseteq y] \implies z \sqsubseteq x \sqcap y\)

- and a bottom element \(\bot:: \quad \exists \ a\)
  \(\bot \sqsubseteq x\)

We often call \(\sqcap\) the **meet** operation.

**Type class:** \('a :: L_{\text{top} \_ \text{bot}}\)
Concretization

We strengthen the abstract interpretation framework by assuming

- \( \text{'av :: } L_{\text{top-bot}} \)
- \( \gamma a_1 \cap \gamma a_2 \subseteq \gamma (a_1 \cap a_2) \)

\[ \implies \gamma (a_1 \cap a_2) = \gamma a_1 \cap \gamma a_2 \]
- \( \implies \cap \) is precise!

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

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

Given $e :: aexp$

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

restrict $S$ to some $S' \subseteq S$ such that

\[
\{ s \in \gamma_f S. \text{ aval } e \text{ s } \in \gamma a \} \subseteq \gamma_f S'
\]

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

What if $\{ s \in \gamma_f S. \text{ aval } e \text{ s } \in \gamma a \}$ is empty?

Work with $'av \text{ st option}$ instead of $'av \text{ st}$.
\text{afilter} \ N

\text{afilter} :: \text{aexp} \Rightarrow \text{'av} \Rightarrow \text{'av st option} \Rightarrow \text{'av st option}

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

An extension of the interface of our framework:
\text{test\_num'} :: \text{int} \Rightarrow \text{'av} \Rightarrow \text{bool}

Assumption:
\text{test\_num'} \ n \ a = (n \in \gamma \ a)

Needed only for computability reasons.
afilter \ V 

\[
\text{afilter } (V \ x) \ a \ S = \\
\begin{align*}
\text{case } S \text{ of } \text{None} & \Rightarrow \text{None} \\
| \text{Some } S & \Rightarrow \\
& \text{let } a' = \text{lookup } S \ x \cap a \\
& \text{in if } a' \sqsubseteq \bot \text{ then None} \\
& \text{else Some (update } S \ x \ a')
\end{align*}
\]
A further extension of the interface of our framework:

\[ \text{filter\_plus'} :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av \]

Assumption:

\[\text{filter\_plus'} a a_1 a_2 = (b_1, b_2) \implies \]
\[\gamma b_1 \supseteq \{ n_1 \in \gamma a_1. \exists n_2 \in \gamma a_2. n_1 + n_2 \in \gamma a \} \land \]
\[\gamma b_2 \supseteq \{ n_2 \in \gamma a_2. \exists n_1 \in \gamma a_1. n_1 + n_2 \in \gamma a \}\]

\[\text{afilter} (\text{Plus} e_1 e_2) a S = \]
\[(\text{let} (b_1, b_2) = \text{filter\_plus'} a (\text{aval}'' e_1 S) (\text{aval}'' e_2 S)\]
\[\text{in} \text{afilter} e_1 b_1 (\text{afilter} e_2 b_2 S))\]

(Alogously for all other arithmetic operations)
Backward analysis of $bexp$

Given $b :: bexp$

$\text{res} :: \text{bool}$ (the intended value of $b$)

$S :: 'av\text{ st option}$

restrict $S$ to some $S' \sqsubseteq S$ such that

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

Roughly: $S'$ overapproximates the subset of $S$ that makes $b$ evaluate to $\text{res}$.
\[ bfilter :: \ bexp \ \Rightarrow \ \text{bool} \ \Rightarrow \ \text{'av st option} \ \Rightarrow \ \text{'av st option} \]
\[
\begin{align*}
\text{bfilter} (Bc \ v) \ \text{res} \ S &= (\text{if } v = \text{res} \ \text{then } S \ \text{else } \text{None}) \\
\text{bfilter} (\text{Not } b) \ \text{res} \ S &= \text{bfilter} b (\neg \ \text{res}) \ S \\
\text{bfilter} (\text{And } b_1 \ b_2) \ \text{res} \ S &= \\
\text{if } \text{res} \ \text{then } \text{bfilter} b_1 \ \text{True} \ (\text{bfilter} b_2 \ \text{True} \ S) \\
\text{else } \text{bfilter} b_1 \ \text{False} \ S \sqcup \ \text{bfilter} b_2 \ \text{False} \ S \\
\end{align*}
\]
\[
\begin{align*}
\text{bfilter} (\text{Less } e_1 \ e_2) \ \text{res} \ S &= \\
\text{let } (\text{res}_1, \ \text{res}_2) &= \\
\quad \text{filter}_{\text{less'}} \ \text{res} \ (\text{aval}'' e_1 \ S) \ (\text{aval}'' e_2 \ S) \\
in \text{afilter} e_1 \ \text{res}_1 \ (\text{afilter} e_2 \ \text{res}_2 \ S) \\
\end{align*}
\]
\( \text{filter\_less'} \) \( \text{res} \ a_1 \ a_2 = (b_1, b_2) \implies \)
\[
\begin{align*}
\gamma \ b_1 & \supseteq \{ n_1 \in \gamma \ a_1. \ \exists \ n_2 \in \gamma \ a_2. \ (n_1 < n_2) = \text{res} \} \land \\
\gamma \ b_2 & \supseteq \{ n_2 \in \gamma \ a_2. \ \exists \ n_1 \in \gamma \ a_1. \ (n_1 < n_2) = \text{res} \}
\end{align*}
\]
\[
\text{step}'
\]

\[
\text{step}'\ S\ (\text{IF } b\ \text{THEN } c_1\ \text{ELSE } c_2\ \{P\}) = \\
\text{IF } b\ \text{THEN } \text{step}'\ (\text{bfilter } b\ \text{True } S)\ c_1 \\
\text{ELSE } \text{step}'\ (\text{bfilter } b\ \text{False } S)\ c_2 \\
\{\text{post } c_1 \sqcap \text{post } c_2\}
\]

\[
\text{step}'\ S\ (\{\text{Inv}\}\ \text{WHILE } b\ \text{DO } c\ \{P\}) = \\
\{S \sqcap \text{post } c\} \\
\text{WHILE } b\ \text{DO } \text{step}'\ (\text{bfilter } b\ \text{True } \text{Inv})\ c \\
\{\text{bfilter } b\ \text{False } \text{Inv}\}
\]
Almost as before, but with correctness lemmas for $afilter$

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

and $bfilter$:

$$\{ s \in \gamma_o S. \ bv = bval \ b \ s \} \subseteq \gamma_o (bfilter \ b \ bv \ S)$$
Extended interface to abstract interpreter:

- `'av :: L_top_bot`
  \[ \gamma \top = \text{UNIV} \quad \text{and} \quad \gamma a_1 \cap \gamma a_2 \subseteq \gamma (a_1 \cap a_2) \]

- `test_num' :: int ⇒ 'av ⇒ bool`
  \[ \text{test}_{\text{num}'} \ n \ a = (n \in \gamma a) \]

- `filter_plus' :: 'av ⇒ 'av ⇒ 'av ⇒ 'av × 'av`
  \[ \text{filter}_{\text{plus}'} (a \ a_1 \ a_2 = (b_1, b_2) ; n_1 \in \gamma a_1 ;
  n_2 \in \gamma a_2 ; n_1 + n_2 \in \gamma a) \]
  \[ \implies n_1 \in \gamma b_1 \land n_2 \in \gamma b_2 \]

- `filter_less' :: bool ⇒ 'av ⇒ 'av ⇒ 'av × 'av`
  \[ \text{filter}_{\text{less}'} (n_1 < n_2) \ a_1 \ a_2 = (b_1, b_2) ;
  n_1 \in \gamma a_1 ; n_2 \in \gamma a_2 \]
  \[ \implies n_1 \in \gamma b_1 \land n_2 \in \gamma b_2 \]
Abs_Int1_ivl.thy
17 Introduction

18 Annotated Commands

19 Collecting Semantics

20 Abstract Interpretation: Orderings

21 A Generic Abstract Interpreter

22 Computable Abstract State

23 Backward Analysis of Boolean Expressions

24 Widening and Narrowing
The Problem

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

Typical example: intervals

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

Can happen even if the program terminates!
Widening — the idea

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

Example: $[l,h_1] \triangledown [l,h_2] = [l,\infty]$ if $h_1 < h_2$
Warning

- $x_{i+1} = f(x_i)$ finds least (post-)fixed point if it terminates and $f$ is monotone
- $x_{i+1} = x_i \nabla f(x_i)$ may return any pfp in the worst case

We win termination, we lose precision
Widening

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

Iterative widening:

\[
\text{while}_{-\text{option}} (\lambda x. \neg f x \sqsubseteq x) (\lambda x. x \nabla f x)
\]

- Correctness (returns pfp): by definition
- Termination: needs more than the two axioms, not covered here

Widening operators can be extended from \('a\) to \('a\ st, 'a\ option\) and \('a\ acom\).
Abs_Int2.thy

Widening
Abstract interpretation with widening

New assumption: \( \text{'av} \) has widening operator

Iterated widening on annotated commands:

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

\[
\text{iter_widen } f = \text{while_option } (\lambda c. \neg f c \sqsubseteq c) (\lambda c. c \bigtriangledown_c f c)
\]

Abstract interpretation of \( c \):

\[
\text{iter_widen } (\text{step'} \top) (\bot_c c)
\]
Interval example

\[x ::= \text{ } N \ 0 \ \{A_0\};\]
\[\{A_1\}\]
\[
\text{WHILE } \text{Less} \ (V \ x) \ (N \ 100) \\
\text{DO } x ::= \text{Plus} \ (V \ x) \ (N \ 1) \ \{A_2\} \\
\{A_3\}\]
Narrowing — the idea

Widening returns a (potentially) imprecise pfp $p$.

If $f$ is monotone, further iteration improves $p$:

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

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

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

Example: interval arithmetic
A narrowing operator \( \triangle :: \text{'a} \Rightarrow \text{'a} \Rightarrow \text{'a} \) must satisfy \( y \sqsubseteq x \Rightarrow y \sqsubseteq x \triangle y \sqsubseteq x \).

**Lemma** Let \( f \) be monotone.

If \( f \ p \sqsubseteq p \sqsubseteq p_0 \) then \( f(p \triangle f p) \sqsubseteq p \triangle f p \sqsubseteq p_0 \)

**Iterative narrowing:**

\[
\text{while\_option} \ (\lambda x. \neg x \sqsubseteq x \triangle f x) \ (\lambda x. x \triangle f x)
\]

- If \( f \) is monotone and we start with a pfp \( p_0 \) of \( f \) and the loop terminates, then (by the lemma) we obtain a pfp of \( f \) below \( p_0 \).
- Termination: not covered here

**Example:** narrowing for intervals
Abstract interpretation with widening & narrowing

New assumption: \( \text{av} \) also has a narrowing operator

\[
\text{iter\_narrow } f = \\
\text{while\_option } (\lambda c. \neg c \sqsubseteq c \triangle_c f c) (\lambda c. c \triangle_c f c)
\]

\[
\text{pfp\_wn } f \ c = \\
(\text{case } \text{iter\_widen } f (\perp_c c) \text{ of } \text{None } \Rightarrow \text{None} \\
| \text{Some } c' \Rightarrow \text{iter\_narrow } f c')
\]

\[
\text{AI\_wn} = \text{pfp\_wn} (\text{step'} \top)
\]