

# Concrete Semantics

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik  
Technische Universität München

2016-2-29

# Part II

## Semantics

# Chapter 7

IMP:

A Simple Imperative Language

- ① IMP Commands
- ② Big-Step Semantics
- ③ Small-Step Semantics

① IMP Commands

② Big-Step Semantics

③ Small-Step Semantics

# Terminology

**Statement:** declaration of fact or claim

*Semantics is easy.*

**Command:** order to do something

*Study the book until you have understood it.*

Expressions are *evaluated*, commands are *executed*

# Commands

Concrete syntax:

$$\begin{array}{l} com ::= \text{SKIP} \\ \quad | \text{ string} ::= aexp \\ \quad | com ; ; com \\ \quad | \text{ IF } bexp \text{ THEN } com \text{ ELSE } com \\ \quad | \text{ WHILE } bexp \text{ DO } com \end{array}$$

# Commands

Abstract syntax:

**datatype** *com* = *SKIP*  
| *Assign string aexp*  
| *Seq com com*  
| *If bexp com com*  
| *While bexp com*



Com.thy

① IMP Commands

② Big-Step Semantics

③ Small-Step Semantics

# Big-step semantics

Concrete syntax:

$$(com, initial-state) \Rightarrow final-state$$

Intended meaning of  $(c, s) \Rightarrow t$ :

Command  $c$  started in state  $s$  terminates in state  $t$

“ $\Rightarrow$ ” here not type!

# Big-step rules

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

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

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

## Big-step rules

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

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

# Big-step rules

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

$$\frac{\begin{array}{c} \text{bval } b \ s_1 \\ (c, s_1) \Rightarrow s_2 \end{array} \quad (\text{WHILE } b \text{ DO } c, s_2) \Rightarrow s_3}{(\text{WHILE } b \text{ DO } c, s_1) \Rightarrow s_3}$$

## Examples: derivation trees

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

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

$$\begin{aligned} \textit{NotEq } a_1\ a_2 &= \\ \textit{Not}(\textit{And } (&\textit{Not}(\textit{Less } a_1\ a_2))\ (\textit{Not}(\textit{Less } a_2\ a_1)))) \end{aligned}$$

Logically speaking

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

is just infix syntax for

$$big\_step\ (c,s)\ t$$

where

$$big\_step :: com \times state \Rightarrow state \Rightarrow bool$$

is an inductively defined predicate.



# Big\_Step.thy

Semantics

# Rule inversion

What can we deduce from

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

# Automating rule inversion

Isabelle command **inductive\_cases** produces theorems that perform rule inversions automatically.

We reformulate the inverted rules. Example:

$$\frac{(c_1;; c_2, s_1) \Rightarrow s_3}{\exists s_2. (c_1, s_1) \Rightarrow s_2 \wedge (c_2, s_2) \Rightarrow s_3}$$

is logically equivalent to

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

Replaces assem  $(c_1;; c_2, s_1) \Rightarrow s_3$  by two assems  $(c_1, s_1) \Rightarrow s_2$  and  $(c_2, s_2) \Rightarrow s_3$  (with a new fixed  $s_2$ ).

No  $\exists$  and  $\wedge$ !

The general format: *elimination rules*

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

(possibly with  $\bigwedge \bar{x}$  in front of the  $asm_i \Longrightarrow P$ )

Reading:

To prove a goal  $P$  with assumption  $asm$ ,  
prove all  $asm_i \Longrightarrow P$

Example:

$$\frac{F \vee G \quad F \Longrightarrow P \quad G \Longrightarrow P}{P}$$

## *elim* attribute

- Theorems with *elim* attribute are used automatically by *blast*, *fastforce* and *auto*
- Can also be added locally, eg (*blast elim: ...*)
- Variant: *elim!* applies elim-rules eagerly.

# Big\_Step.thy

Rule inversion

# Command equivalence

Two commands have the same input/output behaviour:

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

## Example

$$w \sim w'$$

where  $w = \text{WHILE } b \text{ DO } c$

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



# Equivalence proof

$$\begin{aligned} & (w, s) \Rightarrow t \\ & \longleftrightarrow \\ & bval\ b\ s \wedge (\exists s'. (c, s) \Rightarrow s' \wedge (w, s') \Rightarrow t) \\ & \quad \vee \\ & \neg\ bval\ b\ s \wedge t = s \\ & \longleftrightarrow \\ & (w', s) \Rightarrow t \end{aligned}$$

Using the rules and rule inversions for  $\Rightarrow$ .

# Big\_Step.thy

Command equivalence

# Execution is deterministic

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

$$(c, s) \Rightarrow t \implies (c, s) \Rightarrow t' \implies t = t'$$

Proof by rule induction, for arbitrary  $t'$ .

# Big\_Step.thy

Execution is deterministic

# The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

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

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

Big-step semantics cannot directly describe

- nonterminating computations,
- parallel computations.

We need a finer grained semantics!

① IMP Commands

② Big-Step Semantics

③ Small-Step Semantics

# Small-step semantics

Concrete syntax:

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

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

*The first step in the execution of  $c$  in state  $s$  leaves a “remainder” command  $c'$  to be executed in state  $s'$ .*

Execution as finite or infinite reduction:

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



# 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:

$(SKIP, s)$  is final

Why?

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

# Small-step rules

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

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

$$\frac{(c_1, s) \rightarrow (c'_1, s')}{(c_1;; c_2, s) \rightarrow (c'_1;; c_2, s')}$$

## Small-step rules

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

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

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

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

## Small-step examples

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

where  $s = \langle "x'' := 3, "y'' := 7, "z'' := 5 \rangle$ .

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

where

$$\begin{aligned} w &= \text{WHILE } b \text{ DO } c \\ b &= \text{Less } (V "x'') (N 1) \\ c &= "x'' ::= \text{Plus } (V "x'') (N 1) \\ s_n &= \langle "x'' := n \rangle \end{aligned}$$

# Small\_Step.thy

Semantics

Are big and small-step semantics equivalent?

## From $\Rightarrow$ to $\rightarrow^*$

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

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

In two cases a lemma is needed:

**Lemma**

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

Proof by rule induction.



## From $\rightarrow^*$ to $\Rightarrow$

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

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

In the induction step a lemma is needed:

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

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

# Equivalence

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

# Small\_Step.thy

Equivalence of big and small

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

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

By rule inversion:  $(SKIP, s) \rightarrow ct \implies False$

Together:

**Corollary**  $final(c, s) = (c = SKIP)$

# Infinite executions

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

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

Proof:  $(\exists t. cs \Rightarrow t)$   
=  $(\exists t. cs \rightarrow^* (\text{SKIP}, t))$   
    (by big = small)  
=  $(\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$   
    (by final = SKIP)

Equivalent:

$\Rightarrow$  does not yield final state iff  $\rightarrow$  does not terminate

# May versus Must

$\rightarrow$  is deterministic:

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

Therefore: no difference between

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

**must** terminate (all  $\rightarrow$  paths terminate)

Therefore:  $\Rightarrow$  correctly reflects termination behaviour.

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

# Chapter 8

## Compiler



④ Stack Machine

⑤ Compiler

④ Stack Machine

⑤ Compiler

# Stack Machine

Instructions:

**datatype** *instr* =

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

# Semantics

Type synonyms:

$stack = int\ list$

$config = int \times state \times stack$

Execution of 1 instruction:

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

# Instruction execution

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

# Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$instr\ list \vdash config \rightarrow config$$

$$P \vdash c \rightarrow c' =$$

$$(\exists i\ s\ stk.$$

$$c = (i, s, stk) \wedge$$

$$c' = iexec\ (P\ !!\ i)\ (i, s, stk) \wedge$$

$$0 \leq i \wedge i < size\ P)$$

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

# Program execution (\* steps)

Defined in the usual manner:

$$P \vdash (pc, s, stk) \rightarrow^* (pc', s', stk')$$

# Compiler.thy

Stack Machine



④ Stack Machine

⑤ Compiler

## Compiling *aexp*

Same as before:

$$acompile (N\ n) = [LOADI\ n]$$

$$acompile (V\ x) = [LOAD\ x]$$

$$acompile (Plus\ a_1\ a_2) = acompile\ a_1\ @\ acompile\ a_2\ @\ [ADD]$$

Correctness theorem:

*acompile a*

$$\vdash (0, s, stk) \rightarrow^* (size\ (acompile\ a), s, eval\ a\ s \neq stk)$$

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

Needs lemmas!

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

$$\begin{aligned} P \vdash (i, s, stk) \rightarrow^* (i', s', stk') &\implies \\ P' @ P \vdash (size P' + i, s, stk) &\rightarrow^* (size P' + i', s', stk') \end{aligned}$$

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

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

$$\begin{aligned} P \vdash (i, s, stk) \rightarrow (i', s', stk') &\implies \\ P' @ P \vdash (size P' + i, s, stk) &\rightarrow (size P' + i', s', stk') \end{aligned}$$

Proofs by cases.

# Compiling *bexp*

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

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

Principle:

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

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

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

## Example

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

$bcomp \ b \ \text{False} \ \mathcal{B} =$

[ $\text{LOAD} \ "x",$   
 $\text{LOAD} \ "y",$

$\text{LOAD} \ "z",$   
 $\text{LOAD} \ "a",$

]

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

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

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

*bcomp (Less a<sub>1</sub> a<sub>2</sub>) f n =*

*acompa a<sub>1</sub> @*

*acompa a<sub>2</sub> @ (if f then [JMPLESS n] else [JMPGE n])*

*bcomp (And b<sub>1</sub> b<sub>2</sub>) f n =*

*let cb<sub>2</sub> = bcomp b<sub>2</sub> f n;*

*m = if f then size cb<sub>2</sub> else size cb<sub>2</sub> + n;*

*cb<sub>1</sub> = bcomp b<sub>1</sub> False m*

*in cb<sub>1</sub> @ cb<sub>2</sub>*

# Correctness of *bcomp*

$$\begin{aligned} 0 \leq n &\implies \\ bcomp\ b\ f\ n \\ \vdash (0, s, stk) &\rightarrow^* \\ &(\text{size } (bcomp\ b\ f\ n) + (\text{if } f = \text{bval } b\ s \text{ then } n \text{ else } 0), \\ &\quad s, stk) \end{aligned}$$

# Compiling *com*

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

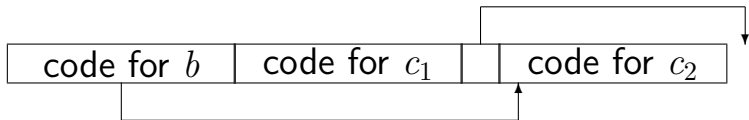
*ccomp* *SKIP* = []

*ccomp* (*x* ::= *a*) = *acom* *a* @ [*STORE* *x*]

*ccomp* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *ccomp* *c*<sub>1</sub> @ *ccomp* *c*<sub>2</sub>

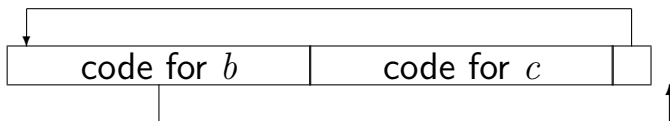


$c_{comp} (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$   
 $let\ cc_1 = c_{comp}\ c_1;\ cc_2 = c_{comp}\ c_2;$   
 $\quad cb = b_{comp}\ b\ False\ (size\ cc_1 + 1)$   
 $in\ cb\ @\ cc_1\ @\ JMP\ (size\ cc_2)\ \# \ cc_2$



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

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



# Correctness of *ccomp*

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

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

Proof by rule induction.

## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*

How about “ $\impliedby$ ”:

*source code simulates compiled code?*

If *ccomp* *c* with start state *s* produces result *t*,  
and if(!)  $(c, s) \Rightarrow t'$ , then “ $\implies$ ” implies  
that *ccomp* *c* with start state *s* must also produce *t'*  
and thus  $t' = t$  (why?).

But we have *not* ruled out this potential error:

*c does not terminate but ccomp c does.*

# The other direction

Two approaches:

- In the absence of nondeterminism:  
Prove that *ccomp* preserves nontermination.  
A nice proof of this fact requires *coinduction*.  
Isabelle supports coinduction, this course avoids it.
- A direct proof: theory *Compiler2*

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

# Chapter 9

## Types

⑥ A Typed Version of IMP

⑦ Security Type Systems

⑥ A Typed Version of IMP

⑦ Security Type Systems



## ⑥ A Typed Version of IMP

Remarks on Type Systems

Typed IMP: Semantics

Typed IMP: Type System

Type Safety of Typed IMP

# Why Types?

*To prevent mistakes, dummy!*

# There are 3 kinds of types

**The Good** Static types that *guarantee* absence of certain runtime faults.

Example: no memory access errors in Java.

**The Bad** Static types that have mostly decorative value but do not guarantee anything at runtime.

Example: C, C++

**The Ugly** Dynamic types that detect errors when it can be too late.

Example: “**TypeError: ...**” in Python.

# The ideal

*Well-typed programs cannot go wrong.*

**Robin Milner**, *A Theory of Type Polymorphism in Programming*, 1978.

The most influential slogan and one of the most influential papers in programming language theory.

# What could go wrong?

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

There are type systems for *everything* (and more) but in practice (Java, C#) only 1 is covered.

# Type safety

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

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

# Correctness and completeness

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

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

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

How about **completeness**? Remember Rice:

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

Hence there is no (decidable) type system that accepts *all* programs that have a certain semantic property.

Automatic analysis of semantic program properties  
is necessarily incomplete.



## ⑥ A Typed Version of IMP

Remarks on Type Systems

**Typed IMP: Semantics**

Typed IMP: Type System

Type Safety of Typed IMP

# Arithmetic

Values:

**datatype**  $val = Iv\ int \mid Rv\ real$

The state:

$state = vname \Rightarrow val$

Arithmetic expressions:

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

# Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!

No automatic coercions.

Does this mean any implementation of IMP also needs to tag values?

No! Compilers compile only well-typed programs, and well-typed programs do not need tags.

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

# Evaluation of *aexp*

Not recursive function but inductive predicate:

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

*taval* (*Ic* *i*) *s* (*Iv* *i*)

*taval* (*Rc* *r*) *s* (*Rv* *r*)

*taval* (*V* *x*) *s* (*s* *x*)

$$\frac{\textit{taval } a_1 \textit{ s (Iv } i_1) \quad \textit{taval } a_2 \textit{ s (Iv } i_2)}{\textit{taval (Plus } a_1 \textit{ } a_2) \textit{ s (Iv (} i_1 + i_2))}}$$
$$\frac{\textit{taval } a_1 \textit{ s (Rv } r_1) \quad \textit{taval } a_2 \textit{ s (Rv } r_2)}{\textit{taval (Plus } a_1 \textit{ } a_2) \textit{ s (Rv (} r_1 + r_2))}}$$

Example: evaluation of  $Plus (V \text{''}x\text{'}) (Ic \ 1)$

If  $s \text{''}x\text{'}' = Iv \ i$ :

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

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

# The functional alternative

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

Exercise!

# Boolean expressions

Syntax as before. Semantics:

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

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

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

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

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

## *com*: big or small steps?

We need to detect if things “go wrong”.

- Big step semantics:  
Cannot model error by absence of final state.  
Would confuse error and nontermination.  
Could introduce an extra error-element, e.g.  
 $\text{big\_step} :: \text{com} \times \text{state} \Rightarrow \text{state option} \Rightarrow \text{bool}$   
Complicates formalization.
- Small step semantics:  
error = semantics gets stuck



# Small step semantics

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

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

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

The other rules remain unchanged.

# Example

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

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

## ⑥ A Typed Version of IMP

Remarks on Type Systems

Typed IMP: Semantics

**Typed IMP: Type System**

Type Safety of Typed IMP

# Type system

There are two types:

**datatype**  $ty = Ity \mid Rty$

What is the type of  $Plus (V \text{"x"}) (V \text{"y"})$  ?

Depends on the type of  $\text{"x"}$  and  $\text{"y"}$  !

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

$tyenv = vname \Rightarrow ty$

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

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

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

# The type of an *aexp*

$$\begin{array}{l} \Gamma \vdash a : \tau \\ \text{tyenv} \vdash \text{aexp} : \text{ty} \end{array}$$

The rules:

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

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

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

$$\frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Plus\ a_1\ a_2 : \tau}$$

## Example

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

where  $\Gamma \text{ ''}x'' = \text{It}y$ .

# Well-typed *bexp*

Notation:

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

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

The rules:

$$\Gamma \vdash Bc\ v$$

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

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

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

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



# Well-typed commands

Notation:

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

Read: *In context  $\Gamma$ ,  $c$  is well-typed.*

The rules:

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

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

$$\frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \textit{IF } b \textit{ THEN } c_1 \textit{ ELSE } c_2}$$

$$\frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash \textit{WHILE } b \textit{ DO } c}$$

# Syntax-directedness

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

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

A syntax-directed set of rules

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

# Syntax-directedness

The big-step semantics is not syntax-directed:

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

## ⑥ A Typed Version of IMP

Remarks on Type Systems

Typed IMP: Semantics

Typed IMP: Type System

Type Safety of Typed IMP

# Well-typed states

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

Remember:

If  $s \Vdash x'' = Rv\ r$

then  $(x'' ::= Plus\ (V\ x'')\ (Ic\ 1),\ s) \not\rightarrow$

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

The type of a value:

$$\textit{type} \ (Iv \ i) = Ity$$

$$\textit{type} \ (Rv \ r) = Rty$$

Well-typed state:

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

# Type soundness

Reduction cannot get stuck:

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

Follows from *progress*:

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

and *preservation*:

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



# The slogan

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

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

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

Preservation: Well-typedness is an *invariant*.

Progress:

$$\llbracket \Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP} \rrbracket \implies \exists cs'. (c, s) \rightarrow cs'$$

Preservation:

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c; \Gamma \vdash s \rrbracket \implies \Gamma \vdash s'$$

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c \rrbracket \implies \Gamma \vdash c'$$

Type soundness:

$$\begin{aligned} &\llbracket (c, s) \rightarrow^* (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq \text{SKIP} \rrbracket \\ &\implies \exists cs''. (c', s') \rightarrow cs'' \end{aligned}$$

*bexp*

Progress:

$$\llbracket \Gamma \vdash b; \Gamma \vdash s \rrbracket \implies \exists v. \text{tbval } b \ s \ v$$

Progress:

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

Preservation:

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

All proofs by rule induction.

Types.thy

# The mantra

Type systems have a purpose:

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

The correctness of the prediction must be provable.

⑥ A Typed Version of IMP

⑦ Security Type Systems



The aim:

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

This is known as *information flow control*.

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

LBS guarantees confidentiality by program analysis,  
not by cryptography.

These analyses are often expressed as type systems.

# Security levels

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

## Two kinds of illicit flows

Explicit: `low := high`

Implicit: `if high1 = high2 then low := 1  
else low := 0`

# Noninterference

*High variables do not interfere with low ones.*

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

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

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

## ⑦ Security Type Systems

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond

# Security Levels

Security levels:

**type\_synonym** *level* = *nat*

Every variable has a security level:

*sec* :: *vname*  $\Rightarrow$  *level*

No definition is needed. Except for examples.

Hence we define (arbitrarily)

*sec* *x* = *length* *x*

## Security Levels on *aexp*

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

*sec* :: *aexp*  $\Rightarrow$  *level*

$$\textit{sec} \ (N \ n) = 0$$

$$\textit{sec} \ (V \ x) = \textit{sec} \ x$$

$$\textit{sec} \ (\textit{Plus} \ a \ b) = \max (\textit{sec} \ a) (\textit{sec} \ b)$$



## Security Levels on *bexp*

*sec* :: *bexp*  $\Rightarrow$  *level*

*sec* (*Bc* *v*) = 0

*sec* (*Not* *b*) = *sec* *b*

*sec* (*And* *b*<sub>1</sub> *b*<sub>2</sub>) = *max* (*sec* *b*<sub>1</sub>) (*sec* *b*<sub>2</sub>)

*sec* (*Less* *a* *b*) = *max* (*sec* *a*) (*sec* *b*)

# Security Levels on States

Agreement of states up to a certain level:

$$s_1 = s_2 (\leq l) \equiv \forall x. \text{sec } x \leq l \longrightarrow s_1 x = s_2 x$$

$$s_1 = s_2 (< l) \equiv \forall x. \text{sec } x < l \longrightarrow s_1 x = s_2 x$$

Noninterference lemmas for expressions:

$$\frac{s_1 = s_2 (\leq l) \quad \text{sec } a \leq l}{\text{aval } a \ s_1 = \text{aval } a \ s_2}$$

$$\frac{s_1 = s_2 (\leq l) \quad \text{sec } b \leq l}{\text{bval } b \ s_1 = \text{bval } b \ s_2}$$

## ⑦ Security Type Systems

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond

# Security Type System

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

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

The well-typedness predicate:

$$l \vdash c$$

Intended meaning:

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

Hence:

“Assignments to variables of level  $< l$  are forbidden.”

# Well-typed or not?

Let  $c =$  *IF Less (V "x1") (V "x")*  
*THEN "x1" ::= N 0*  
*ELSE "x1" ::= N 1*

$1 \vdash c$  ?      Yes

$2 \vdash c$  ?      Yes

$3 \vdash c$  ?      No

# The type system

$$l \vdash \textit{SKIP}$$

$$\frac{\textit{sec } a \leq \textit{sec } x \quad l \leq \textit{sec } x}{l \vdash x ::= a}$$

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

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

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

Remark:

$l \vdash c$  is syntax-directed and executable.

# Anti-monotonicity

$$\frac{l \vdash c \quad l' \leq l}{l' \vdash c}$$

Proof by ... as usual.

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



# Confinement

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

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

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

Proof by ... as usual.

# Noninterference

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

Proof by ... as usual.

## ⑦ Security Type Systems

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond

The  $l \vdash c$  system is intuitive and executable

- but in the literature a more elegant formulation is dominant
- which does not need *max*
- and works for arbitrary partial orders.

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

$$\frac{l \vdash' c \quad l' \leq l}{l' \vdash' c}$$

together with one rule per construct:

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

$$\frac{\text{sec } a \leq \text{sec } x \quad l \leq \text{sec } x}{l \vdash' x ::= a}$$

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

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

$$\frac{\text{sec } b \leq l \quad l \vdash' c}{l \vdash' \text{WHILE } b \text{ DO } c}$$

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

## Equivalence of $\vdash$ and $\vdash'$

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

Proof by induction.

Use subsumption directly below *IF* and *WHILE*.

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

Proof by induction. Subsumption already a lemma for  $\vdash$ .

## ⑦ Security Type Systems

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond



- Systems  $l \vdash c$  and  $l \vdash' c$  are *top-down*:  
level  $l$  comes from the context  
and is checked at  $::=$  commands.
- System  $\vdash c : l$  is *bottom-up*:  
 $l$  is the minimal level of any variable assigned in  $c$   
and is checked at *IF* and *WHILE* commands.

$$\vdash \textit{SKIP} : l$$

$$\frac{\textit{sec } a \leq \textit{sec } x}{\vdash x ::= a : \textit{sec } x}$$

$$\frac{\vdash c_1 : l_1 \quad \vdash c_2 : l_2}{\vdash c_1;; c_2 : \textit{min } l_1 \ l_2}$$

$$\frac{\textit{sec } b \leq \textit{min } l_1 \ l_2 \quad \vdash c_1 : l_1 \quad \vdash c_2 : l_2}{\vdash \textit{IF } b \textit{ THEN } c_1 \textit{ ELSE } c_2 : \textit{min } l_1 \ l_2}$$

$$\frac{\textit{sec } b \leq l \quad \vdash c : l}{\vdash \textit{WHILE } b \textit{ DO } c : l}$$

## Equivalence of $\vdash :$ and $\vdash'$

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

Proof by induction.

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

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

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

Proof by induction.

## ⑦ Security Type Systems

Secure IMP

A Security Type System

A Type System with Subsumption

A Bottom-Up Type System

Beyond

Does noninterference really guarantee  
absence of information flow?

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

Beware of covert channels!

*$0 \vdash \text{WHILE Less } (V \text{ "x"}) \text{ (N 1) DO SKIP}$*

A drastic solution:

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

New typing rule:

$$\frac{sec\ b = 0 \quad 0 \vdash c}{0 \vdash WHILE\ b\ DO\ c}$$

Now provable:

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

# Further extensions

- Time
- Probability
- Quantitative analysis
- More programming language features:
  - exceptions
  - concurrency
  - OO
  - ...

# Literature

The inventors of security type systems are Volpano and Smith.

For an excellent survey see

Sabelfeld and Myers. *Language-Based Information-Flow Security*. 2003.



# Chapter 10

## Data-Flow Analyses and Optimization

⑧ Definite Initialization Analysis

⑨ Live Variable Analysis

⑧ Definite Initialization Analysis

⑨ Live Variable Analysis

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

## Java Language Specification

Java was the first language to force programmers to initialize their variables.

## Examples: ok or not?

Assume  $x$  is initialized:

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

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

Assume  $x$  and  $y$  are initialized:

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

# Simplifying principle

*We do not analyze boolean expressions to determine program execution.*

## ⑧ Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics

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

*vars* :: *aexp*  $\Rightarrow$  *vname set*

*vars* (*N* *n*) = {}

*vars* (*V* *x*) = {*x*}

*vars* (*Plus* *a*<sub>1</sub> *a*<sub>2</sub>) = *vars* *a*<sub>1</sub>  $\cup$  *vars* *a*<sub>2</sub>

*vars* :: *bexp*  $\Rightarrow$  *vname set*

*vars* (*Bc* *v*) = {}

*vars* (*Not* *b*) = *vars* *b*

*vars* (*And* *b*<sub>1</sub> *b*<sub>2</sub>) = *vars* *b*<sub>1</sub>  $\cup$  *vars* *b*<sub>2</sub>

*vars* (*Less* *a*<sub>1</sub> *a*<sub>2</sub>) = *vars* *a*<sub>1</sub>  $\cup$  *vars* *a*<sub>2</sub>



Vars.thy

## ⑧ Definite Initialization Analysis

Prelude: Variables in Expressions

**Definite Initialization Analysis**

Initialization Sensitive Semantics

Modified example from the JLS:

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

Similar statements for each language construct.

$D :: \text{vname set} \Rightarrow \text{com} \Rightarrow \text{vname set} \Rightarrow \text{bool}$

$D A c A'$  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.*

$$\begin{array}{c}
D \ A \ SKIP \ A \\
\hline
vars \ a \subseteq A \\
\hline
D \ A \ (x ::= a) \ (insert \ x \ A) \\
\hline
D \ A_1 \ c_1 \ A_2 \quad D \ A_2 \ c_2 \ A_3 \\
\hline
D \ A_1 \ (c_1;; c_2) \ A_3 \\
\hline
vars \ b \subseteq A \quad D \ A \ c_1 \ A_1 \quad D \ A \ c_2 \ A_2 \\
\hline
D \ A \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ (A_1 \cap A_2) \\
\hline
vars \ b \subseteq A \quad D \ A \ c \ A' \\
\hline
D \ A \ (WHILE \ b \ DO \ c) \ A
\end{array}$$

## Correctness of $D$

- Things can go wrong:  
execution may access uninitialized variable.  
 $\implies$  We need a new, finer-grained semantics.
- Big step semantics:  
semantics longer, correctness proof shorter
- Small step semantics:  
semantics shorter, correctness proof longer

For variety's sake, we choose a big step semantics.

## ⑧ Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics

*state = vname  $\Rightarrow$  val option*

where

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

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

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



# Expression evaluation

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

*aval (N i) s = Some i*

*aval (V x) s = s x*

*aval (Plus a<sub>1</sub> a<sub>2</sub>) s =  
(case (aval a<sub>1</sub> s, aval a<sub>2</sub> s) of  
  (Some i<sub>1</sub>, Some i<sub>2</sub>)  $\Rightarrow$  Some(i<sub>1</sub>+i<sub>2</sub>)  
  | \_  $\Rightarrow$  None)*

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

*bval (Bc v) s = Some v*

*bval (Not b) s =*

*(case bval b s of None  $\Rightarrow$  None*  
*| Some bv  $\Rightarrow$  Some ( $\neg$  bv))*

*bval (And b<sub>1</sub> b<sub>2</sub>) s =*

*(case (bval b<sub>1</sub> s, bval b<sub>2</sub> s) of*  
*(Some bv<sub>1</sub>, Some bv<sub>2</sub>)  $\Rightarrow$  Some(bv<sub>1</sub>  $\wedge$  bv<sub>2</sub>)*  
*| \_  $\Rightarrow$  None)*

*bval (Less a<sub>1</sub> a<sub>2</sub>) s =*

*(case (aval a<sub>1</sub> s, aval a<sub>2</sub> s) of*  
*(Some i<sub>1</sub>, Some i<sub>2</sub>)  $\Rightarrow$  Some(i<sub>1</sub> < i<sub>2</sub>)*  
*| \_  $\Rightarrow$  None)*

# Big step semantics

$$(com, state) \Rightarrow state\ option$$

A small complication:

$$\frac{(c_1, s_1) \Rightarrow Some\ s_2 \quad (c_2, s_2) \Rightarrow s}{(c_1;; c_2, s_1) \Rightarrow s}$$
$$\frac{(c_1, s_1) \Rightarrow None}{(c_1;; c_2, s_1) \Rightarrow None}$$

More convenient, because compositional:

$$(com, state\ option) \Rightarrow state\ option$$

Error (*None*) propagates:

$$(c, \textit{None}) \Rightarrow \textit{None}$$

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

$$(\textit{SKIP}, s) \Rightarrow s$$

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

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

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

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

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

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

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

$$\frac{\begin{array}{l} bval\ b\ s = Some\ True \\ (c,\ Some\ s) \Rightarrow s' \quad (WHILE\ b\ DO\ c,\ s') \Rightarrow s'' \end{array}}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

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

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

We want in the end:

*Well-initialized programs cannot go wrong.*

*If  $D(\text{dom } s) \subseteq A'$  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  $D A \subseteq A'$  and  $A \subseteq \text{dom } s$   
then  $\exists t. s' = \text{Some } t \wedge A' \subseteq \text{dom } t$ .*

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

Proof needs some easy lemmas:

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

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

$$D \ A \ c \ A' \implies A \subseteq A'$$



⑧ Definite Initialization Analysis

⑨ Live Variable Analysis

# Motivation

Consider the following program:

```
x := y + 1;  
y := y + 2;  
x := y + 3
```

The first assignment is redundant and can be removed because  $x$  is **dead** at that point.

Semantically, a variable  $x$  is live before command  $c$  if the initial value of  $x$  can influence the final state.

A weaker but easier to check condition:

We call  $x$  *live* before  $c$

if there is some potential execution of  $c$

where  $x$  is read before it can be overwritten.

Implicitly, every variable is read at the end of  $c$ .

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

$x := 0$  ☹️

$y := x; y := 0; x := 0$  😊

WHILE  $b$  DO  $y := x; x := 1$  😊

At the end of a command, we may be interested in the value of *only some of the variables*, e.g. *only the global variables* at the end of a procedure.

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

# Liveness analysis

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

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

$L\ SKIP\ X = X$

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

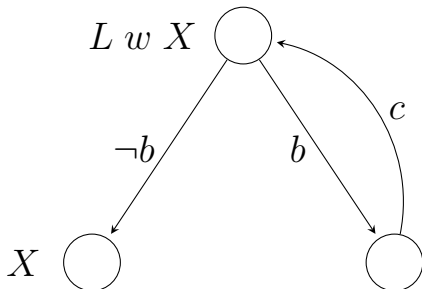
$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

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

Example:

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

# WHILE $b$ DO $c$



$L w X$  must satisfy

$vars\ b \quad \subseteq \quad L w X$  (evaluation of  $b$ )

$X \quad \subseteq \quad L w X$  (exit)

$L\ c\ (L w X) \quad \subseteq \quad L w X$  (execution of  $c$ )

We define

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

$\implies$

$$\textit{vars } b \subseteq L \ w \ X \quad \checkmark$$

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

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

$$\begin{aligned}
L \text{ SKIP } X &= X \\
L (x ::= a) X &= \text{vars } a \cup (X - \{x\}) \\
L (c_1;; c_2) X &= L c_1 (L c_2 X) \\
L (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) X &= \text{vars } b \cup L c_1 X \cup L c_2 X \\
L (\text{WHILE } b \text{ DO } c) X &= \text{vars } b \cup X \cup L c X
\end{aligned}$$

Example:

$$\begin{aligned}
L (\text{WHILE Less } (V \text{ "x"}) (V \text{ "x"}) \text{ DO "y" } ::= V \text{ "z"}) \\
\{\text{"x"}\} &= \{\text{"x"}, \text{"z"}\}
\end{aligned}$$



# Gen/kill analyses

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

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

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

$$\begin{aligned} X_1 \subseteq X_2 &\implies A \ c \ X_1 \subseteq A \ c \ X_2 \\ A \ c \ (X_1 \cap X_2) &= A \ c \ X_1 \cap A \ c \ X_2 \end{aligned}$$

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

# Liveness via gen/kill

*kill :: com  $\Rightarrow$  vname set*

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

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP* =  $\{\}$

*gen* (*x* ::= *a*) = *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *gen c*<sub>1</sub>  $\cup$  (*gen c*<sub>2</sub> − *kill c*<sub>1</sub>)

*gen* (*IF* *b* *THEN* *c*<sub>1</sub> *ELSE* *c*<sub>2</sub>) =  
*vars b*  $\cup$  *gen c*<sub>1</sub>  $\cup$  *gen c*<sub>2</sub>

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

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

Proof by induction on  $c$ .

$\implies$

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

## Digression: definite initialization via *gen/kill*

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

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

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

## ⑨ Live Variable Analysis

Correctness of  $L$

Dead Variable Elimination

True Liveness

Comparisons

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

*The value of the final state on  $X$   
only depends on  
the value of the initial state on  $L \subset X$ .*

Put differently:

*If two initial states agree on  $L \subset X$   
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*):

$$s_1 = s_2 \text{ on vars } a \implies \text{aval } a\ s_1 = \text{aval } a\ s_2$$

$$s_1 = s_2 \text{ on vars } b \implies \text{bval } b\ s_1 = \text{bval } b\ s_2$$



## Correctness of $L$

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

Proof by rule induction.

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

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

## Optimality of $L w$

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

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

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

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

## ⑨ Live Variable Analysis

Correctness of  $L$

Dead Variable Elimination

True Liveness

Comparisons

Bury all assignments to dead variables:

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

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if*  $x \in X$  *then*  $x ::= a$  *else* *SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* = *bury*  $c_1$  (*L*  $c_2$  *X*);; *bury*  $c_2$  *X*

*bury (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) X* =  
*IF*  $b$  *THEN* *bury*  $c_1$  *X* *ELSE* *bury*  $c_2$  *X*

*bury (WHILE b DO c) X* =  
*WHILE*  $b$  *DO* *bury*  $c$  (*L* (*WHILE*  $b$  *DO*  $c$ ) *X*)

# Correctness of *bury*

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

where *UNIV* is the set of all variables.

The two directions need to be proved separately.

$$(c, s) \Rightarrow s' \implies (\text{bury } c \text{ } UNIV, s) \Rightarrow s'$$

Follows from generalized statement:

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

Proof by rule induction, like for correctness of  $L$ .

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

Follows from generalized statement:

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

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

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

$$(\exists c_1\ c_2.$$

$$c = c_1;; c_2 \wedge$$

$$bc_2 = bury\ c_2\ X \wedge bc_1 = bury\ c_1\ (L\ c_2\ X))$$

## ⑨ Live Variable Analysis

Correctness of  $L$

Dead Variable Elimination

True Liveness

Comparisons



# Terminology

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

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

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

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

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

## Application to $L w$

Remember the specification of  $L w$ :

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

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

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

and in particular of  $L\ c$ .

## True liveness

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

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

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

Better:

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

But then

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

is not correct anymore.

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

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

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

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

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

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

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

$$c = y ::= V \ x;; x ::= V \ z$$

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

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

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

$L \ w$  for true liveness

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

# Existence of least fixpoints

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

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

is the least pre-fixpoint and least fixpoint of  $f$ .

# Proof of Knaster-Tarski

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

**Proof**

- $f(lfp f) \subseteq lfp f$
- $lfp f$  is the least pre-fixpoint of  $f$

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

**Proof**

- $f p \leq p \implies f p = p$
- $p$  is the least fixpoint



## Definition of $L$

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

$L (WHILE\ b\ DO\ c) X = lfp\ f_w$   
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  $lfp$  is monotone:

$lfp\ f \subseteq lfp\ g$  if for all  $X$ ,  $f\ X \subseteq g\ X$

**Corollary**  $f_w$  is monotone.

# Computation of $lfp$

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

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

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

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

## Computation of $lfp f_w$

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

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

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

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

**Proof** by induction on  $c$

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

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

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

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

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

## Example

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

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

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

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

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

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

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

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

# Computation of *lfp* in Isabelle

From the library theory `While_Combinator`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$   
 $while\ b\ f\ s = (if\ b\ s\ then\ while\ b\ f\ (f\ s)\ else\ s)$

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

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

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

# Limiting the number of iterations

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

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

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

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

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

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

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

## ⑨ Live Variable Analysis

Correctness of  $L$

Dead Variable Elimination

True Liveness

Comparisons

# Comparison of analyses

- Definite initialization analysis is a *forward must analysis*:
  - it analyses the executions starting from some point,
  - variables *must* be assigned (on every program path) before they are used.
- Live variable analysis is a *backward may analysis*:
  - it analyses the executions ending in some point,
  - live variables *may* be used (on some program path) before they are assigned.



# Comparison of DFA frameworks

Program representation:

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

# Chapter 11

## Denotational Semantics

10 A Relational Denotational Semantics of IMP

11 Continuity

# What is it?

A denotational semantics maps syntax to semantics:

$$D :: \textit{syntax} \Rightarrow \textit{meaning}$$

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

$D$  must be defined by primitive recursion over the syntax

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

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

# Why?

More abstract:

operational:   How to execute it

denotational:   What does it mean

Simpler proof principles:

operational:   relational, rule induction

denotational:   equational, structural induction

## 10 A Relational Denotational Semantics of IMP

## 11 Continuity

# Relations

$Id :: ('a \times 'a) \text{ set}$

$Id = \{p. \exists x. p = (x, x)\}$

$op \ O :: ('a \times 'b) \text{ set} \Rightarrow ('b \times 'c) \text{ set} \Rightarrow ('a \times 'c) \text{ set}$

$r \ O \ s = \{(x, z). \exists y. (x, y) \in r \wedge (y, z) \in s\}$

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

**type\_synonym** *com\_den* = (*state*  $\times$  *state*) *set*

$$D \text{ SKIP} = Id$$

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

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

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



## Example

Let  $c_1 = \text{"}x\text{"} ::= N\ 0$   
 $c_2 = \text{"}y\text{"} ::= V\ \text{"}x\text{"}.$

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

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

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

$$D (WHILE\ b\ DO\ c) = ?$$

Wanted:

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

**Problem:** not a denotational definition  
not allowed by Isabelle

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

General principle:

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

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

*W*

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

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

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

**Lemma**  $W\ db\ dc$  is monotone.

We define

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

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

$$\begin{aligned} D\ w &= lfp\ f = f\ (lfp\ f) = W\ (bval\ v)\ (D\ c)\ (D\ w) \\ &= \{(s, t). \text{ if } bval\ b\ s \text{ then } (s, t) \in D\ c\ O\ D\ w \text{ else } s = t\} \end{aligned}$$

## Why least?

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

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

Then

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

Every relation is a fixpoint!

Only the least relation  $\{\}$  makes computational sense.

# A denotational equivalence proof

## Example

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

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

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

$$\begin{aligned} D\ w \\ &= \{(s, t). \text{ if } bval\ b\ s \text{ then } (s, t) \in D\ c\ O\ D\ w \text{ else } s = t\} \\ &= D\ (IF\ b\ THEN\ c;;\ w\ ELSE\ SKIP) \end{aligned}$$

# Equivalence of denotational and big-step semantics

**Lemma**  $(c, s) \Rightarrow t \implies (s, t) \in D\ c$

**Proof** by rule induction

**Lemma**  $(s, t) \in D\ c \implies (s, t) \in \textit{Big\_step}\ c$

**Proof** by induction on  $c$

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

10 A Relational Denotational Semantics of IMP

11 Continuity



# Chains and continuity

## Definition

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

## Definition (Continuous)

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

**Lemma**  $cont\ f \Longrightarrow mono\ f$

# Kleene fixpoint theorem

**Theorem**  $\text{cont } f \implies \text{lfp } f = (\bigcup_n f^n \{\})$

# Application to semantics

**Lemma**  $W \ db \ dc$  is continuous.

## Example

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

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

Let  $f = W \ db \ dc$

where  $db = bval \ b = (\lambda s. s \text{ "x" } \neq 0)$

$dc = D \ c = \{(s, t). t = s(\text{"x"} := s \text{ "x" } - 1)\}$

# A proof of determinism

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

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

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

# Chapter 12

## Hoare Logic

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

## 12 Partial Correctness

### Introduction

The Syntactic Approach

The Semantic Approach

Soundness and Completeness



We have proved functional programs correct  
(e.g. a compiler).

We have proved properties of imperative languages  
(e.g. type safety).

But how do we prove properties of imperative programs?

An example program:

*"y" ::= N 0;; wsum*

where

*wsum*  $\equiv$

*WHILE Less (N 0) (V "x")*

*DO ("y" ::= Plus (V "y") (V "x"));*

*"x" ::= Plus (V "x") (N (- 1)))*

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

*sum i = (if  $i \leq 0$  then 0 else sum (i - 1) + i)*

# A proof via operational semantics

Theorem:

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

Required Lemma:

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

Proved by rule induction.

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

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

But no free lunch:

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

## 12 Partial Correctness

Introduction

**The Syntactic Approach**

The Semantic Approach

Soundness and Completeness

This is the standard approach.

Formulas are syntactic objects.

Everything is very concrete and simple.

But complex to formalize.

Hence we soon move to a semantic view of formulas.

Reason for introduction of syntactic approach: didactic

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

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

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

Informal example:

$$\{x = 41\} \ x := x + 1 \ \{x = 42\}$$

Terminology:  $P$  and  $Q$  are called *assertions*.

# Examples

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

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

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

Boundary cases:

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

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

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



# The rules of Hoare Logic

$$\{P\} \text{ SKIP } \{P\}$$

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

Notation:  $Q[a/x]$  means “ $Q$  with  $a$  substituted for  $x$ ”.

Examples:

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

Intuitive explanation of backward-looking rule:

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

Afterwards we can replace all occurrences of  $a$  in  $Q$  by  $x$ .

The assignment axiom allows us  
to compute the precondition from the postcondition.

There is a version to compute the postcondition from  
the precondition, but it is more complicated. (Exercise!)

## More rules of Hoare Logic

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

$$\frac{\{P \wedge b\} \ c_1 \ \{Q\} \quad \{P \wedge \neg b\} \ c_2 \ \{Q\}}{\{P\} \ \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \{Q\}}$$

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

In the While-rule,  $P$  is called an *invariant* because it is preserved across executions of the loop body.

# The *consequence* rule

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

$$\frac{P' \longrightarrow P \quad \{P\} \text{ c } \{Q\} \quad Q \longrightarrow Q'}{\{P'\} \text{ c } \{Q'\}}$$

*Preconditions can be strengthened,  
postconditions can be weakened.*

## Two derived rules

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

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

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

# Example

$\{x = i\}$

$y := 0;$

*WHILE*  $0 < x$  *DO* ( $y := y+x; x := x-1$ )

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

Example proof exhibits key properties of Hoare logic:

- Choice of rules is syntax-directed and hence automatic.
- Proof of “;” proceeds from right to left.
- Proofs require only invariants and arithmetic reasoning.

## 12 Partial Correctness

Introduction

The Syntactic Approach

**The Semantic Approach**

Soundness and Completeness



Assertions are predicates on states

$$assn = state \Rightarrow bool$$

Alternative view: *sets of states*

Semantic approach simplifies meta-theory, our main objective.

# Validity

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

$$\longleftrightarrow$$

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

“ $\{P\} \text{ c } \{Q\}$  is valid”

In contrast:

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

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

# Provability

$$\vdash \{P\} \text{ SKIP } \{P\}$$

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

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

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

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

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

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

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

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

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

Hoare\_Examples.thy

## 12 Partial Correctness

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:  $\models \Rightarrow \vdash$

# 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 $wp$

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

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

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

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

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

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

# Completeness

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

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

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

**Proof** by induction on  $c$ , for arbitrary  $Q$ .

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

**Fact**  $\models \{P\} \ c \ \{Q\} \iff (\forall s. P \ s \longrightarrow wp \ c \ Q \ s)$

Follows directly from defs of  $\models$  and  $wp$ .

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

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

## WARNING

Most texts that discuss completeness of Hoare logic state or prove that Hoare logic is only “relatively complete” but **not complete**.

Reason: the standard notion of completeness assumes some abstract mathematical notion of  $\models$ .

Our notion of  $\models$  is defined within the same (limited) proof system (for HOL) as  $\vdash$ .

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

Idea:

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

More precisely:

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

Method:

*Simulate syntax-directed application of Hoare logic rules. Collect all assertion language side conditions.*



# A problem: loop invariants

Where do they come from?

A trivial solution:

Let the user provide them!

How?

Each loop must be annotated with its invariant!

How to synthesize loop invariants automatically  
is an important research problem.

Which we ignore for the moment.

But come back to later.

Terminology:

VCG = Verification Condition Generator

All successful verification technology for imperative programs relies on

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

# The (approx.) plan of attack

- 1 Introduce **annotated** commands with loop invariants
- 2 Define functions for *computing*
  - weakest preconditions:  $pre :: com \Rightarrow assn \Rightarrow assn$
  - verification conditions:  $vc :: com \Rightarrow assn \Rightarrow bool$
- 3 Soundness:  $vc\ c\ Q \implies \vdash \{ ? \}\ c\ \{ Q \}$
- 4 Completeness: if  $\vdash \{ P \}\ c\ \{ Q \}$  then  $c$  can be annotated (becoming  $C$ ) such that  $vc\ C\ Q$ .

The details are a bit different ...

# Annotated commands

Like commands, except for *While*:

**datatype** *acom* = *Askip*  
                  | *Aassign vname aexp*  
                  | *Aseq acom acom*  
                  | *Aif bexp acom acom*  
                  | *Awhile assn bexp acom*

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

*{I} WHILE b DO c*

# Weakest precondition

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

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

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

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

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

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

## Warning

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

# Verification condition

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

$vc \text{ SKIP } Q = True$

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

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

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

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



Verification conditions only arise from loops:

- the invariant must be invariant
- and it must imply the postcondition.

Everything else in the definition of *vc* is just bureaucracy: collecting assertions and passing them around.

Hoare triples operate on *com*,  
functions *pre* and *vc* operate on *acom*.  
Therefore we define

$$\textit{strip} :: \textit{acom} \Rightarrow \textit{com}$$

$$\textit{strip} \textit{SKIP} = \textit{SKIP}$$

$$\textit{strip} (x ::= a) = x ::= a$$

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

$$\textit{strip} (\textit{IF } b \textit{ THEN } C_1 \textit{ ELSE } C_2) = \\ \textit{IF } b \textit{ THEN } \textit{strip} C_1 \textit{ ELSE } \textit{strip} C_2$$

$$\textit{strip} (\{I\} \textit{ WHILE } b \textit{ DO } C) = \textit{WHILE } b \textit{ DO } \textit{strip} C$$

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

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

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

Corollary:

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

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

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

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

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

Why could premise not be provable although conclusion is?

- Some annotation in  $C$  is not invariant.
- $vc$  or  $pre$  are wrong  
(e.g. accidentally always produce *False*).

Therefore we prove completeness:  
suitable annotations exist such that premise is provable.

# Completeness of *vc* & *pre* w.r.t. $\vdash$

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

Proof by rule induction. Needs two monotonicity lemmas:

$$\llbracket \forall s. P \ s \longrightarrow P' \ s; \ pre \ C \ P \ s \rrbracket \implies pre \ C \ P' \ s$$

$$\llbracket \forall s. P \ s \longrightarrow P' \ s; \ vc \ C \ P \rrbracket \implies vc \ C \ P'$$

12 Partial Correctness

13 Verification Conditions

14 Total Correctness

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

Total Correctness = Partial Correctness + Termination

Formally:

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

Assumes that semantics is deterministic!

Exercise: Reformulate for nondeterministic language

# $\vdash_t$ : A proof system for total correctness

Only need to change the *WHILE* rule.

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

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



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

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

# Hoare\_Total.thy

Example

# Soundness

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

Proof by induction, with a nested induction on  $n$  in the While-case.

# Completeness

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

Follows easily from

$$\vdash_t \{wp_t \ c \ Q\} \ c \ \{Q\}$$

where

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

Proof of  $\vdash_t \{wp_t \ c \ Q\} \ c \ \{Q\}$  is by induction on  $c$ .

In the *WHILE*  $b \ DO \ c$  case, use the *WHILE* rule with

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

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

# Chapter 13

## Abstract Interpretation

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

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



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

# Applications: Optimization

- Constant folding
- Unreachable and dead code elimination
- Array access optimization:

$a[i] := 1; a[j] := 2; x := a[i] \rightsquigarrow$

$a[i] := 1; a[j] := 2; x := 1$

if  $i \neq j$

- ...

# Applications: Debugging/Verification

Detect presence or absence of certain runtime exceptions/errors:

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

# Precision

A consequence of Rice's theorem:

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

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

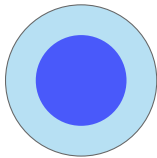
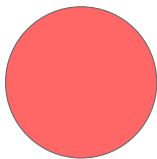
If an analysis says that some value

- cannot arise, this is definitely the case.
- can arise, this is only potentially the case.

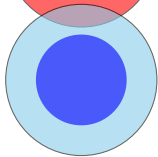
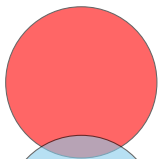
Beware of *false alarms* because of overapproximation.

Error

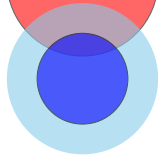
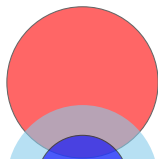
Program  
Analysis



No Alarm



False Alarm



True Alarm

# Annotated commands

Like in Hoare logic, we annotate

$$\{ \dots \}$$

program text with semantic information.

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

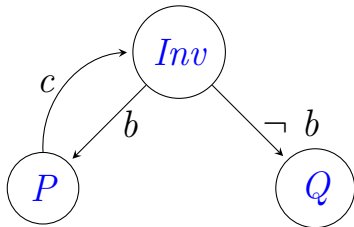
$$x := 0 \{ \dots \}; y := 0 \{ \dots \}$$

# Annotated WHILE

View

$$\begin{array}{l} \{Inv\} \\ \text{WHILE } b \text{ DO } \{P\} \ c \\ \{Q\} \end{array}$$

as a *control flow graph* with annotated nodes:



# The starting point: Collecting Semantics

Collects all possible states for each program point:

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



Infinite sets of states:

$$\{ \dots, \langle x := -1 \rangle, \langle x := 0 \rangle, \langle x := 1 \rangle, \dots \}$$

WHILE  $x < 3$

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

$x := x+2 \{ \dots, \langle x := 3 \rangle, \langle x := 4 \rangle \}$   
 $\{ \langle x := 3 \rangle, \langle x := 4 \rangle, \dots \}$

Multiple variables:

```
x := 0; y := 0 { <x:=0, y:=0> } ;  
{ <x:=0, y:=0>, <x:=2, y:=1>, <x:=4, y:=2> }  
WHILE x < 3  
DO { <x:=0, y:=0>, <x:=2, y:=1> }  
    x := x+2; y := y+1  
    { <x:=2, y:=1>, <x:=4, y:=2> }  
{ <x:=4, y:=2> }
```

# A first approximation

$$(vname \Rightarrow val) \text{ set} \quad \rightsquigarrow \quad vname \Rightarrow val \text{ set}$$

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

Loses relationships between variables  
but simplifies matters a lot.

Example:

$\{ \langle x:=0, y:=0 \rangle, \langle x:=1, y:=1 \rangle \}$

is approximated by

$\langle x:=\{0,1\}, y:=\{0,1\} \rangle$

which also subsumes

$\langle x:=0, y:=1 \rangle$  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

## Example

Consistently annotated program:

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

The annotations are computed by

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

```
x := 0
```

```
WHILE x < 3
```

```
DO
```

```
    x := x+2
```

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



# Concrete syntax

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

*'a*: type of annotations

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

# Abstract syntax

## **datatype**

*'a acom =*

*SKIP 'a*

| *Assign string aexp 'a*

| *Seq ('a acom) ('a acom)*

| *If bexp 'a ('a acom) 'a ('a acom) 'a*

| *While 'a bexp 'a ('a acom) 'a*

## Auxiliary functions: *strip*

Strips all annotations from an annotated command

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

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

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

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

$$\begin{aligned} \textit{strip} (\textit{IF} b \textit{ THEN} \{P_1\} C_1 \textit{ ELSE} \{P_2\} C_2 \{P\}) \\ = \textit{IF} b \textit{ THEN} \textit{strip} C_1 \textit{ ELSE} \textit{strip} C_2 \end{aligned}$$

$$\begin{aligned} \textit{strip} (\{I\} \textit{ WHILE} b \textit{ DO} \{P\} C \{Q\}) \\ = \textit{WHILE} b \textit{ DO} \textit{strip} C \end{aligned}$$

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

## Auxiliary functions: *annos*

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

$$\textit{annos} :: 'a \textit{ acom} \Rightarrow 'a \textit{ list}$$

$$\textit{annos} (\textit{SKIP} \{P\}) = [P]$$

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

$$\textit{annos} (C_1;; C_2) = \textit{annos} C_1 @ \textit{annos} C_2$$

$$\begin{aligned} \textit{annos} (\textit{IF } b \textit{ THEN } \{P_1\} C_1 \textit{ ELSE } \{P_2\} C_2 \{Q\}) = \\ P_1 \# \textit{annos} C_1 @ P_2 \# \textit{annos} C_2 @ [Q] \end{aligned}$$

$$\begin{aligned} \textit{annos} (\{I\} \textit{ WHILE } b \textit{ DO } \{P\} C \{Q\}) = \\ I \# P \# \textit{annos} C @ [Q] \end{aligned}$$

## Auxiliary functions: *anno*

$anno :: 'a \text{ acom} \Rightarrow \text{nat} \Rightarrow 'a$

$anno\ C\ p = annos\ C\ !\ p$

The  $p$ -th annotation (starting from 0)

## Auxiliary functions: *post*

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

$post \ C = last \ (annos \ C)$

The rightmost/last/post annotation

## Auxiliary functions: *map\_acom*

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

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

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



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

The annotations are generated iteratively:

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

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

*start states*

flowing into the command

*step*

$$\text{step } S (\text{SKIP } \{-\}) = \text{SKIP } \{S\}$$

$$\begin{aligned} \text{step } S (x ::= e \{-\}) = \\ x ::= e \{ \{s(x := \text{aval } e \ s) \mid s. s \in S\} \} \end{aligned}$$

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

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

*step*

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

# Collecting semantics

View command as a control flow graph

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

Stabilization means fixpoint:

$$\textit{step } S \ C = C$$

Collecting\_Examples.thy

## Abstract example

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

*step*  $S$   $C = C$  means

$$I = S \cup \text{post } C_0$$

$$P = \{ s \in I. \text{ bval } b \ s \}$$

$$C_0 = \text{step } P \ C_0$$

$$Q = \{ s \in I. \neg \text{ bval } b \ s \}$$

**Fixpoint = solution of equation system**

Iteration is just one way of solving equations

## Why *least* fixpoint?

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

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

But the “reachable” fixpoint is  $I = \{\}$

Does *step* always have a least fixpoint?



# Partial order

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

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

# Complete lattice

## Definition

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

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

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

**Fact** Type  $\langle a \rangle$  set is a complete lattice where  $\leq = \subseteq$  and  $\sqcap = \bigcap$

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

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

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

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

# Existence of least fixpoints

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

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

$$\sqcap \{p. f p \leq p\}.$$

**Proof** just like the version for sets.

## Ordering '*a* *acom*

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

$$\begin{aligned} C_1 \leq C_2 &\iff \\ \text{strip } C_1 &= \text{strip } C_2 \wedge \\ (\forall p < \text{length } (\text{annos } C_1). &\text{ anno } C_1 \ p \leq \text{ anno } C_2 \ p) \end{aligned}$$

**Lemma** If '*a* is a partial order, so is '*a* *acom*.

# Ordering '*a acom*

Example:

$$x ::= N\ 0\ \{\{a\}\} \leq x ::= N\ 0\ \{\{a, b\}\} \iff \textit{True}$$

$$x ::= N\ 0\ \{\{a\}\} \leq x ::= N\ 0\ \{\{\}\} \iff \textit{False}$$

$$x ::= N\ 0\ \{S\} \leq x ::= N\ 1\ \{S\} \iff \textit{False}$$

The collecting semantics needs to order *state set acom*.

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

Does *state set acom* also form a complete lattice?

Almost ...

## A complication

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

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

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

Only *strip*-equal commands have an infimum



It turns out:

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

Therefore we make the carrier set explicit.

## Complete lattice as a set

**Definition** Let  $'a$  be a partially ordered type.

A set  $L :: 'a$  set is a *complete lattice*

if every  $M \subseteq L$  has a greatest lower bound  $\bigwedge M \in L$ .

Given sets  $A$  and  $B$  and a function  $f$ ,

$f \in A \rightarrow B$  means  $\forall a \in A. f\ a \in B$ .

**Theorem** (Knaster-Tarski)

Let  $L :: 'a$  set be a complete lattice

and  $f \in L \rightarrow L$  a monotone function.

Then  $f$  (restricted to  $L$ ) has the least fixpoint

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

## Application to *acom*

Let  $'a$  be a complete lattice and  $c :: com$ .

Then  $L = \{C :: 'a \text{ acom. strip } C = c\}$

is a complete lattice.

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

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

Example  $\sqcap \{SKIP \{A\}, SKIP \{B\}, \dots\}$   
 $= SKIP \{\sqcap \{A, B, \dots\}\}$

Formally ...

## Auxiliary function: *annotate*

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

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

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

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

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

**Proof** straightforward (pointwise).

# The Collecting Semantics

The underlying complete lattice is now *state set*.

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

**Lemma** *step*  $S \in L \rightarrow L$  and is monotone.

Therefore Knaster-Tarski is applicable and we define

$CS :: \text{com} \Rightarrow \text{state set acom}$

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

[*lfp* is defined in the context of some lattice  $L$ .

Our concrete  $L$  depends on  $c$ .

Therefore *lfp* depends on  $c$ , too.]

# Relationship to big-step semantics

For simplicity: compare only pre and post-states

**Theorem**  $(c, s) \Rightarrow t \implies t \in \text{post}(CS\ c)$

Follows directly from

$$\llbracket (c, s) \Rightarrow t; s \in S \rrbracket \implies t \in \text{post}(\text{lfp } c \ (\text{step } S))$$

Proof of

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

uses

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

and

$$\begin{aligned} &\llbracket (c, s) \Rightarrow t; \text{strip } C = c; s \in S; \text{step } S \text{ } C \leq C \rrbracket \\ &\Longrightarrow t \in \text{post } C \end{aligned}$$

which is proved by induction on the big step.



In a nutshell:

collecting semantics overapproximates big-step semantics

Later:

program analysis overapproximates collecting semantics

Together:

program analysis overapproximates big-step semantics

The other direction

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

is also true but is not proved in this course.

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

# Approximating the Collecting semantics

A conceptual step:

$$(vname \Rightarrow val) \text{ set} \quad \rightsquigarrow \quad vname \Rightarrow val \text{ set}$$

A domain-specific step:

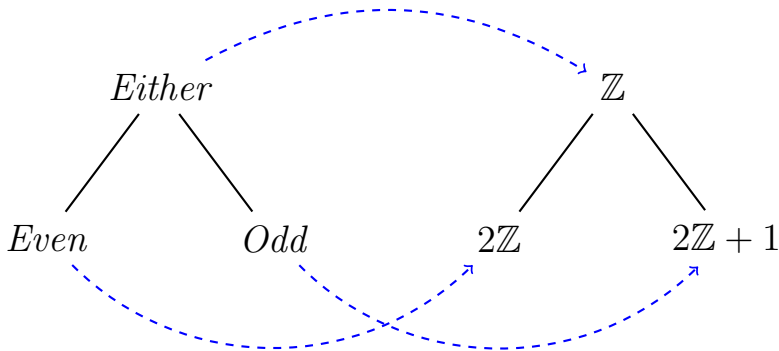
$$val \text{ set} \quad \rightsquigarrow \quad 'av$$

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

## Example: parity analysis

Abstract values:

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



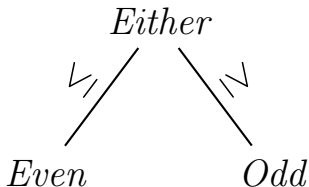
concretization function  $\gamma_{parity}$

*A concretisation function  $\gamma$*

maps an abstract value to a set of concrete values

Bigger abstract values represent more concrete values

## Example: parity



**Fact** Type *parity* is a partial order.

# Top element

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

$$a \leq \top$$

# Semilattice

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

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

$$\sqcup :: 'a \Rightarrow 'a \Rightarrow 'a$$

$$x \leq x \sqcup y \quad y \leq x \sqcup y$$

$$\llbracket x \leq z; y \leq z \rrbracket \implies x \sqcup y \leq z$$

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

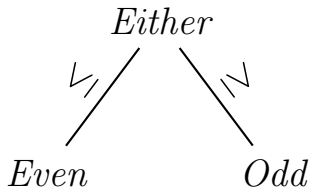


$\leq$  uniquely determines  $\square$

**Fact** If  $\langle A, \leq \rangle$  is a semilattice, then the least upper bound of two elements is uniquely determined.

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

## Example: parity



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

# Isabelle's type classes

A *type class* is defined by

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

Examples

class *order*: partial orders

class *semilattice\_sup*: semilattices

class *semilattice\_sup\_top*: semilattices with top element

A type belongs to some class if

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

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

Example:  $\text{parity} :: \text{semilattice\_sup}$

HOL/Orderings.thy  
Abs\_Int1\_parity.thy

Orderings and instances

# From abstract values to abstract states

Need to abstract collecting semantics:

*state set*

- First attempt:

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

where  $'av$  is the type of abstract values

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

# Lifting semilattice and $\gamma$ to *'av st option*

**Lemma** If *'a :: semilattice\_sup\_top*  
then *'b  $\Rightarrow$  'a :: semilattice\_sup\_top*

**Proof**

$$\begin{aligned}(f \leq g) &= (\forall x. f\ x \leq g\ x) \\ f \sqcup g &= (\lambda x. f\ x \sqcup g\ x) \\ \top &= (\lambda x. \top)\end{aligned}$$

**definition**

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

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

**Lemma**    *If 'a :: semilattice\_sup\_top*  
              *then 'a option :: semilattice\_sup\_top*

**Proof**

$$(Some\ x \leq Some\ y) = (x \leq y)$$

$$(None \leq \_) = True$$

$$(Some\ \_ \leq None) = False$$

$$Some\ x \sqcup Some\ y = Some\ (x \sqcup y)$$

$$None \sqcup y = y$$

$$x \sqcup None = x$$

$$\top = Some\ \top$$

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

**fun**  $\gamma\_option :: ('a \Rightarrow 'c\ set) \Rightarrow 'a\ option \Rightarrow 'c\ set$   
**where**

$\gamma\_option\ \gamma\ None = \{\}$

$\gamma\_option\ \gamma\ (Some\ a) = \gamma\ a$

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



*'a acom*

Remember:

**Lemma** If *'a :: order* then *'a acom :: order*.

Partial order is enough, semilattice not needed.

Lifting  $\gamma :: 'a \Rightarrow 'c$  to  $'a\ acom \Rightarrow 'c\ acom$  is easy:  
*map\_acom*

**Lemma**

If  $\gamma$  is monotone then *map\_acom*  $\gamma$  is monotone.

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

- Stepwise development of a **generic abstract interpreter** as a parameterized module
- Parameters/Input: abstract type of values together with abstractions of the operations on concrete type  $val = int$ .
- 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  $'av :: \textit{semilattice\_sup\_top}$

Concretization function:  $\gamma :: 'av \Rightarrow \textit{val set}$

Assumptions:  $a \leq b \implies \gamma a \subseteq \gamma b$   
 $\gamma \top = \textit{UNIV}$

## Parameters (II)

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

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

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

Assumptions:

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

$$\llbracket i_1 \in \gamma a_1; i_2 \in \gamma a_2 \rrbracket \implies i_1 + i_2 \in \gamma (plus' a_1 a_2)$$

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

# Lifted concretization functions

$\gamma_s :: 'av\ st \Rightarrow state\ set$

$\gamma_s = \gamma\_fun\ \gamma$

$\gamma_o :: 'av\ st\ option \Rightarrow state\ set$

$\gamma_o = \gamma\_option\ \gamma_s$

$\gamma_c :: 'a\ st\ option\ acom \Rightarrow state\ set\ acom$

$\gamma_c = map\_acom\ \gamma_o$

All of them are monotone.

# Abstract interpretation of *aexp*

**fun** *aval'* :: *aexp*  $\Rightarrow$  'av st  $\Rightarrow$  'av

*aval'* (*N* *n*) *S* = *num'* *n*

*aval'* (*V* *x*) *S* = *S* *x*

*aval'* (*Plus* *a*<sub>1</sub> *a*<sub>2</sub>) *S* = *plus'* (*aval'* *a*<sub>1</sub> *S*) (*aval'* *a*<sub>2</sub> *S*)

Correctness of *aval'* wrt *aval*:

**Lemma**  $s \in \gamma_s S \implies \text{aval } a \ s \in \gamma (\text{aval}' a \ S)$

**Proof** by induction on *a*

using the assumptions about the parameters.

## Example instantiation with *parity*

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

*num\_parity*  $i = (\text{if } i \bmod 2 = 0 \text{ then Even else Odd})$

*plus\_parity* Even Even = Even

*plus\_parity* Odd Odd = Even

*plus\_parity* Even Odd = Odd

*plus\_parity* Odd Even = Odd

*plus\_parity* Either  $y = \text{Either}$

*plus\_parity*  $x \text{ Either} = \text{Either}$



## Example instantiation with *parity*

Input:  $\gamma \mapsto \gamma_{\text{parity}}$   
 $\text{num}' \mapsto \text{num\_parity}$   
 $\text{plus}' \mapsto \text{plus\_parity}$

Must prove parameter assumptions

Output:  $\text{aval}' \mapsto \text{aval\_parity}$

Example The value of

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

is *Even*.

Abs\_Int1\_parity.thy

Locale interpretation

# Abstract interpretation of *bexp*

For now, boolean expressions are not analysed.

# Abstract interpretation of *com*

Abstracting the collecting semantics

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

to

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

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

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

*Step* :: 'a  $\Rightarrow$  'a acom  $\Rightarrow$  'a acom

Parameterized wrt

- type 'a with  $\sqcup$
- the interpretation of assignments and tests:

*asem* :: vname  $\Rightarrow$  aexp  $\Rightarrow$  'a  $\Rightarrow$  'a

*bsem* :: bexp  $\Rightarrow$  'a  $\Rightarrow$  'a

$$\text{Step } a \text{ (SKIP } \{-\}) = \text{SKIP } \{a\}$$

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

$$\text{Step } a \text{ (} C_1;; C_2 \text{)} = \text{Step } a \text{ } C_1;; \text{Step (post } C_1 \text{)} C_2$$

$$\begin{aligned} \text{Step } a \text{ (IF } b \text{ THEN } \{P_1\} C_1 \text{ ELSE } \{P_2\} C_2 \{-\}) = \\ \text{IF } b \text{ THEN } \{b\text{sem } b \text{ } a\} \text{Step } P_1 C_1 \\ \text{ELSE } \{b\text{sem (Not } b \text{)} a\} \text{Step } P_2 C_2 \\ \{ \text{post } C_1 \sqcup \text{post } C_2 \} \end{aligned}$$

$$\begin{aligned} \text{Step } a \text{ (} \{I\} \text{ WHILE } b \text{ DO } \{P\} C \{-\}) = \\ \{a \sqcup \text{post } C\} \text{ WHILE } b \text{ DO } \{b\text{sem } b \text{ } I\} \text{Step } P C \\ \{b\text{sem (Not } b \text{)} I\} \end{aligned}$$

## Instantiating *Step*

The truth: *asem* and *bsem* are (hidden) parameters of *Step*: *Step asem bsem ...*

$$\begin{aligned} \textit{step} = \\ \textit{Step} \ (\lambda x \ e \ S. \ \{s(x := \textit{aval} \ e \ s) \mid s. \ s \in S\}) \\ (\lambda b \ S. \ \{s \in S. \ \textit{bval} \ b \ s\}) \end{aligned}$$
$$\textit{step}' = \textit{Step} \ \textit{asem} \ (\lambda b \ S. \ S)$$

where

$$\begin{aligned} \textit{asem} \ x \ e \ S = \\ (\textit{case} \ S \ \textit{of} \ \textit{None} \Rightarrow \textit{None} \\ \mid \textit{Some} \ S \Rightarrow \textit{Some} \ (S(x := \textit{aval}' \ e \ S))) \end{aligned}$$

## Example: iterating *step\_parity*

$$(\textit{step\_parity } S)^k c$$

where

$$\begin{aligned} c = & \quad x ::= N \ 3 \ \{None\} \ ; \\ & \quad \{None\} \\ & \quad WHILE \ b \ DO \ \{None\} \\ & \quad \quad x ::= Plus \ (V \ x) \ (N \ 5) \ \{None\} \\ & \quad \{None\} \end{aligned}$$

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

$$S_p = \quad Some \ ((\lambda_. \ Either)(x := p))$$



## Correctness of $step'$ wrt $step$

The concretization of  $step'$  overapproximates  $step$ :

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

where  $S :: 'av\ st\ option$

$C :: 'av\ st\ option\ acom$

**Lemma**  $Step\ f\ g\ (\gamma_o S) (\gamma_c C) \leq \gamma_c (Step\ f'\ g'\ S\ C)$

if for all  $x, e, b$ :  $f\ x\ e\ (\gamma_o S) \subseteq \gamma_o (f'\ x\ e\ S)$

$g\ b\ (\gamma_o S) \subseteq \gamma_o (g'\ b\ S)$

**Proof** by an easy induction on  $C$

# The abstract interpreter

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

# Unbounded search

From the HOL library:

*while\_option ::*

*$('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ option}$*

such that

*while\_option b f x =*

*(if b x then while\_option b f (f x) else Some x)*

and *while\_option b f x = None*

if the recursion does not terminate.

Pre-fixpoint:

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

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

Start iteration with least annotated command:

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

# The generic abstract interpreter

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

**Theorem**  $AI\ c = Some\ C \implies CS\ c \leq \gamma_c\ C$

**Proof** From the assumption:  $step' \top\ C \leq C$

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

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

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

Because  $CS$  is the least pfp of  $step\ UNIV$ :  $CS\ c \leq \gamma_c\ C$

# Problem

*AI* is not directly executable

because *pf<sub>p</sub>* compares  $f\ C \leq C$

where  $C :: 'av\ st\ option\ acom$

which compares functions  $vname \Rightarrow 'av$

which is not computable: *vname* is infinite.

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

# Solution

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

An association list representation:

**type\_synonym** *'a st\_rep* = (*vname*  $\times$  *'a*) *list*

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

**fun** *fun\_rep* :: (*'a::top*) *st\_rep*  $\Rightarrow$  (*vname*  $\Rightarrow$  *'a*)

*fun\_rep* ((*x*, *a*) # *ps*) = (*fun\_rep* *ps*)(*x* := *a*)

*fun\_rep* [] = ( $\lambda x. \top$ )

Missing variables are mapped to  $\top$

Example: *fun\_rep* [(*"x"*, *a*), (*"x"*, *b*)]

= (( $\lambda x. \top$ )(*"x"* := *b*))(*"x"* := *a*) = ( $\lambda x. \top$ )(*"x"* := *a*)



## Comparing association lists

Compare them only on their finite “domains”:

$$\begin{aligned} less\_eq\_st\_rep\ ps_1\ ps_2 = \\ (\forall x \in set\ (map\ fst\ ps_1) \cup set\ (map\ fst\ ps_2). \\ \quad fun\_rep\ ps_1\ x \leq fun\_rep\ ps_2\ x) \end{aligned}$$

Not a partial order because not antisymmetric!

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

## Quotient type $'a\ st$

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

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

**quotient\_type**  $'a\ st = ('a::top)\ st\_rep\ /\ eq\_st$

Elements of  $'a\ st$ :

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

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

## Alternative to quotient: canonical representatives

For example, the subtype of **sorted** association lists:

- $[("x", a), ("y", b)]$
- $[("y", b), ("x", a)]$

More concrete, and probably a bit more complicated

## Auxiliary functions on $'a\ st$

Turning an abstract state into a function:

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

Updating an abstract state:

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

## Turning 'a st into a semilattice

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

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

**fun** map2\_st\_rep ::

$$('a::top \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ st\_rep \Rightarrow 'a\ st\_rep \Rightarrow 'a\ st\_rep$$

$$map2\_st\_rep\ f\ ((x, y) \# ps_1)\ ps_2 =$$

$$let\ y_2 = fun\_rep\ ps_2\ x$$

$$in\ (x, f\ y\ y_2) \# map2\_st\_rep\ f\ ps_1\ ps_2$$

$$map2\_st\_rep\ f\ []\ ps_2 = map\ (\lambda(x, y). (x, f\ \top\ y))\ ps_2$$

# Modified abstract interpreter

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

$$S\ x \quad \rightsquigarrow \quad fun\ S\ x$$

$$S(x := a) \rightsquigarrow update\ S\ x\ a$$

Now  $AI$  is executable!

Abs\_Int1\_parity.thy  
Abs\_Int1\_const.thy

Examples

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



# Beyond partial correctness

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

The solution: **Monotonicity**



**Precision** *AI* computes *least* pre-fixpoints

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

# Monotonicity

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

$$\llbracket a_1 \leq b_1; a_2 \leq b_2 \rrbracket \implies \text{plus}' a_1 a_2 \leq \text{plus}' b_1 b_2$$

**Theorem** In the monotone framework,  $\text{aval}'$  is also monotone

$$S_1 \leq S_2 \implies \text{aval}' e S_1 \leq \text{aval}' e S_2$$

and therefore  $\text{step}'$  is also monotone:

$$\llbracket S_1 \leq S_2; C_1 \leq C_2 \rrbracket \implies \text{step}' S_1 C_1 \leq \text{step}' S_2 C_2$$

# Precision: smaller is better

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

**Lemma** Let  $\leq$  be a partial order on a set  $L$  with **least element**  $\perp \in L$ :  $x \in L \implies \perp \sqsubseteq x$ .

Let  $f \in L \rightarrow L$  be a **monotone function**.

If *while\_option*  $(\lambda x. \neg f x \leq x)$   $f \perp = \text{Some } p$  then  $p$  is the least pre-fixpoint of  $f$  on  $L$ .

That is, if  $f q \leq q$  for some  $q \in L$ , then  $p \leq q$ .

**Proof** Clearly  $f p \leq p$ .

Given any pre-fixpoint  $q \in L$ , property

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

is an invariant of the while loop:

$$P \perp \text{ holds and } P x \text{ implies } f x \leq f q \leq q$$

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

Application to

$$\begin{aligned} AI\ c &= pfp\ (step'\ \top)\ (bot\ c) \\ pfp\ f &= while\_option\ (\lambda x. \neg f\ x \leq x)\ f \end{aligned}$$

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

# Termination

Because  $step'$  is monotone, starting from *bot*  $c$  generates an ascending  $<$  chain of annotated commands.

We exhibit a measure function  $m_c$  that decreases with every loop iteration:

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

Modulo some details ...

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

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

Assumptions:  $m\ x \leq h$   
 $x < y \implies m\ y < m\ x$

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

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

# Measure functions

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

$$m_c\ C = (\sum a \leftarrow annos\ C. m_o\ a\ (vars\ C))$$

$$m_o :: 'av\ st\ option \Rightarrow vname\ set \Rightarrow nat$$

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

$$m_o\ None\ X = h * card\ X + 1$$

$$m_s :: 'av\ st \Rightarrow vname\ set \Rightarrow nat$$

$$m_s\ S\ X = (\sum x \in X. m\ (S\ x))$$



All measure functions are bounded:

$$\text{finite } X \implies m_s S X \leq h * \text{card } X$$

$$\text{finite } X \implies m_o \text{opt } X \leq h * \text{card } X + 1$$

$$m_c C \leq \text{length } (\text{annos } C) * (h * \text{card } (\text{vars } C) + 1)$$

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

# Complication

Anti-monotonicity does not hold!

Example:

$$\text{finite } X \implies S_1 < S_2 \implies m_s S_2 X < m_s S_1 X$$

$$\text{because } S_1 < S_2 \iff S_1 \leq S_2 \wedge (\exists x. S_1 x < S_2 x)$$

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

## *top\_on*

$top\_on_s :: 'av\ st \Rightarrow\ vname\ set \Rightarrow\ bool$

$top\_on_s\ S\ X = (\forall x \in X. S\ x = \top)$

$top\_on_o :: 'av\ st\ option \Rightarrow\ vname\ set \Rightarrow\ bool$

$top\_on_o\ (Some\ S)\ X = top\_on_s\ S\ X$

$top\_on_o\ None\ X = True$

$top\_on_c :: 'av\ st\ option\ acom \Rightarrow\ bool$

$top\_on_c\ C\ X = (\forall a \in set\ (annos\ C). top\_on_o\ a\ X)$

Now we can formulate and prove anti-monotonicity:

$$\begin{aligned} & \llbracket \textit{finite } X; S_1 = S_2 \textit{ on } - X; S_1 < S_2 \rrbracket \\ \implies & m_s S_2 X < m_s S_1 X \end{aligned}$$

$$\begin{aligned} & \llbracket \textit{finite } X; \textit{top\_on}_o o_1 (- X); \textit{top\_on}_o o_2 (- X); \\ & \quad o_1 < o_2 \rrbracket \\ \implies & m_o o_2 X < m_o o_1 X \end{aligned}$$

$$\begin{aligned} & \llbracket \textit{top\_on}_c C_1 (- \textit{vars } C_1); \textit{top\_on}_c C_2 (- \textit{vars } C_2); \\ & \quad C_1 < C_2 \rrbracket \\ \implies & m_c C_2 < m_c C_1 \end{aligned}$$

Now we can prove termination

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

because  $step'$  leaves  $top\_on_s$  invariant:

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

Warning: *step'* is very inefficient.

It is applied to every subcommand in every step.

Thus the actual complexity of *AI* is  $O(p^2 * n * h)$

Better iteration policy:

Ignore subcommands where nothing has changed.

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

As usual: efficiency complicates proofs.

Abs\_Int1\_parity.thy  
Abs\_Int1\_const.thy

Termination

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



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

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

Given  $S :: \text{'av st}$ , reduce it to some  $S' \leq S$  such that

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

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

Computing  $S'$  from  $S$  requires  $\sqcap$

# Lattice

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

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

$$\sqcap :: 'a \Rightarrow 'a \Rightarrow 'a$$

$$x \sqcap y \leq x \quad x \sqcap y \leq y$$

$$\llbracket z \leq x; z \leq y \rrbracket \implies z \leq x \sqcap y$$

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

Type class: *lattice*

# Bounded lattice

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

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

Type class: *bounded\_lattice*

**Fact** Any complete lattice is a bounded lattice.

# Concretization

We strengthen the abstract interpretation framework by assuming

- $\text{'av} :: \textit{bounded\_lattice}$
- $\gamma \ a_1 \cap \gamma \ a_2 \subseteq \gamma \ (a_1 \sqcap a_2)$   
 $\implies \gamma \ (a_1 \sqcap a_2) = \gamma \ a_1 \cap \gamma \ a_2$   
 $\implies \sqcap$  is precise!

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

- $\gamma \ \perp = \{\}$

# Backward analysis of *aexp*

Given  $e :: aexp$

$a :: 'av$  (the intended value of  $e$ )

$S :: 'av\ st$

restrict  $S$  to some  $S' \leq S$  such that

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

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

What if  $\{s \in \gamma_s\ S. \text{aval } e\ s \in \gamma\ a\}$  is empty?

Work with *'av st option* instead of *'av st*

*inv\_aval' N*

*inv\_aval' ::*

*aexp  $\Rightarrow$  'av  $\Rightarrow$  'av st option  $\Rightarrow$  'av st option*

*inv\_aval' (N n) a S =*

*(if test\_num' n a then S else None)*

An extension of the interface of our framework:

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

Assumption:

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

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

*inv\_aval' V*

```
inv_aval' (V x) a S =  
case S of None => None  
| Some S =>  
  let a' = fun S x => a  
  in if a' = ⊥ then None  
    else Some (update S x a')
```

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

## *inv\_aval' Plus*

A further extension of the interface of our framework:

$$\text{inv\_plus}' :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$$

Assumption:

$$\begin{aligned} \text{inv\_plus}' a a_1 a_2 = (a_1', a_2') \implies \\ \gamma a_1' \supseteq \{i_1 \in \gamma a_1. \exists i_2 \in \gamma a_2. i_1 + i_2 \in \gamma a\} \wedge \\ \gamma a_2' \supseteq \{i_2 \in \gamma a_2. \exists i_1 \in \gamma a_1. i_1 + i_2 \in \gamma a\} \end{aligned}$$

Definition:

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

(Analogously for all other arithmetic operations)



# Backward analysis of *bexp*

Given  $b :: bexp$

$res :: bool$  (the intended value of  $b$ )

$S :: 'av\ st\ option$

restrict  $S$  to some  $S' \leq S$  such that

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

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

*inv\_bval'* ::

*bexp*  $\Rightarrow$  *bool*  $\Rightarrow$  'av st option  $\Rightarrow$  'av st option

*inv\_bval'* (*Bc v*) *res S* = (*if v = res then S else None*)

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

*inv\_bval'* (*And b<sub>1</sub> b<sub>2</sub>*) *res S* =

*if res*

*then inv\_bval'* *b<sub>1</sub>* *True* (*inv\_bval'* *b<sub>2</sub>* *True S*)

*else inv\_bval'* *b<sub>1</sub>* *False S*  $\sqcup$  *inv\_bval'* *b<sub>2</sub>* *False S*

*inv\_bval'* (*Less e<sub>1</sub> e<sub>2</sub>*) *res S* =

*let* (*a<sub>1</sub>*, *a<sub>2</sub>*) = *inv\_less'* *res* (*aval'' e<sub>1</sub> S*) (*aval'' e<sub>2</sub> S*)

*in inv\_aval'* *e<sub>1</sub>* *a<sub>1</sub>* (*inv\_aval'* *e<sub>2</sub>* *a<sub>2</sub>* *S*)

A further extension of the interface of our framework:

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

Assumption:

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

## Example: intervals, informally

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

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

$$\text{inv\_bval}' (\mathbf{x} + \mathbf{y} < \mathbf{z}) \text{True}$$

$$\{\mathbf{x} \mapsto [10, 20], \mathbf{y} \mapsto [-10, 0], \mathbf{z} \mapsto [-5, 5]\}:$$

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

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

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

$$\text{inv\_aval}' \mathbf{x} [10, 14] \{\bullet\} =$$

$$\{\mathbf{x} \mapsto [10, 14], \mathbf{y} \mapsto [-10, -6], \mathbf{z} \mapsto [1, 5]\}$$

*step'*

Before:  $step' = Step\ asem\ (\lambda b\ S.\ S)$

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

# Correctness proof

Almost as before, but with correctness lemmas for  $inv\_aval'$

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

and  $inv\_bval'$ :

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

# Summary

Extended interface to abstract interpreter:

- $'av :: \text{bounded\_lattice}$   
 $\gamma \perp = \{\}$  and  $\gamma a_1 \cap \gamma a_2 \subseteq \gamma (a_1 \sqcap a_2)$
- $\text{test\_num}' :: \text{int} \Rightarrow 'av \Rightarrow \text{bool}$   
 $\text{test\_num}' i a = (i \in \gamma a)$
- $\text{inv\_plus}' :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$   
 $\llbracket \text{inv\_plus}' a a_1 a_2 = (a_1', a_2') \rrbracket$   
 $i_1 \in \gamma a_1; i_2 \in \gamma a_2; i_1 + i_2 \in \gamma a$   
 $\implies i_1 \in \gamma a_1' \wedge i_2 \in \gamma a_2'$
- $\text{inv\_less}' :: \text{bool} \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$   
 $\llbracket \text{inv\_less}' (i_1 < i_2) a_1 a_2 = (a_1', a_2') \rrbracket$   
 $i_1 \in \gamma a_1; i_2 \in \gamma a_2$   
 $\implies i_1 \in \gamma a_1' \wedge i_2 \in \gamma a_2'$

Abs\_Int2\_ivl.thy



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

# The problem

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

Canonical example: intervals

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

Can happen even if the program terminates!

# Widening

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

Example: Widening on (non-empty) intervals

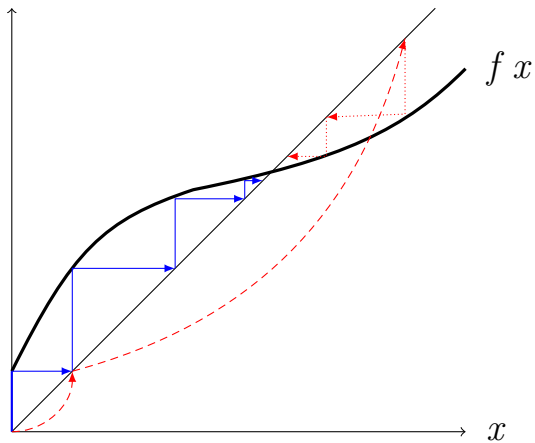
$$[l_1, h_1] \nabla [l_2, h_2] = [l, h]$$

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

## Warning

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

We win termination, we lose precision



- iteration of  $f$
- - - iteration with widening
- ⋯ narrowing iteration of  $f$

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

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

# Abstract interpretation with widening

New assumption:  $'av$  has widening operator

$iter\_widen :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ option}$

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

Correctness (returns pfp): by definition

Abstract interpretation of  $c$ :

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



# Interval example

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

# Narrowing

Widening returns a (potentially) **imprecise** pfp  $p$ .

If  $f$  is **monotone**, further iteration improves  $p$ :

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

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

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

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

**Lemma** Let  $f$  be monotone.

If  $f\ p \leq p$  then  $f(p \triangle f\ p) \leq p \triangle f\ p \leq p$

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

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

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

Example: Narrowing on (non-empty) intervals

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

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

# Abstract interpretation with widening & narrowing

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

$$\begin{aligned} pfp\_wn\ f\ x = \\ (\text{case } iter\_widen\ f\ x\ \text{of } None \Rightarrow None \\ \mid Some\ p \Rightarrow iter\_narrow\ f\ p) \end{aligned}$$
$$AI\_wn\ c = pfp\_wn\ (step'\ \top)\ (bot\ c)$$

**Theorem**  $AI\_wn\ c = Some\ C \implies CS\ c \leq \gamma_c\ C$

**Proof** as before

# Termination

of

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

via measure function  $m$

such that  $m$  goes down with every iteration:

$$P\ x \implies m\ x > m(g\ x)$$

May need some invariant  $Inv$  as additional premise:

$$Inv\ x \implies P\ x \implies m\ x > m(g\ x)$$

## Termination of *iter\_widen*

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

As before (almost): Assume  $m :: 'av \Rightarrow nat$  and  $h :: nat$   
such that  $m\ x \leq h$  and  $x \leq y \Longrightarrow m\ y \leq m\ x$   
and additionally  $\neg y \leq x \Longrightarrow m\ (x \nabla y) < m\ x$

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

Termination of *iter\_widen* on *'a st option acom*:

**Lemma**  $\neg C_2 \leq C_1 \Longrightarrow m_c\ (C_1 \nabla C_2) < m_c\ C_1$   
if *top\_on<sub>c</sub>*  $C_1$  ( $\neg vars\ C_1$ ), *top\_on<sub>c</sub>*  $C_2$  ( $\neg vars\ C_2$ )  
and *strip*  $C_1 = strip\ C_2$

# Termination of *iter\_narrow*

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

Assume  $n :: 'av \Rightarrow nat$  such that

$$\llbracket y \leq x; x \triangle y < x \rrbracket \Longrightarrow n (x \triangle y) < n x$$

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

Termination of *iter\_narrow* on *'a st option acom*:

**Lemma**  $\llbracket C_2 \leq C_1; C_1 \triangle C_2 < C_1 \rrbracket \Longrightarrow$   
 $n_c (C_1 \triangle C_2) < n_c C_1$  if *strip*  $C_1 = \text{strip } C_2$ ,  
*top\_on<sub>c</sub>*  $C_1$  ( $- \text{vars } C_1$ ) and *top\_on<sub>c</sub>*  $C_2$  ( $- \text{vars } C_2$ )



# Measuring non-empty intervals

$$m[l, h] = (\text{if } l = -\infty \text{ then } 0 \text{ else } 1) + \\ (\text{if } h = \infty \text{ then } 0 \text{ else } 1)$$

$$h = 2$$

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