Semantics with Applications: Model-Based Program Analysis

©Hanne Riis Nielson ©Flemming Nielson

Computer Science Department, Aarhus University, Denmark

(October 1996)

Contents

Preface

These notes on model-based program analysis have been used for teaching semantics-based program analysis to third year students familiar with denotational semantics as covered in Chapter 4 of

> H.R.Nielson, F.Nielson:
> *Semantics with Applications: A Formal Introduction*,
> Wiley, 1992. [ISBN 0 471 92980 8]

(referred to as [NN] in the sequel). Indeed, these notes may be used as an alternative to the treatment of static program analysis in Chapter 5 of [NN]. The present notes go deeper into the ideas behind program analysis: how to define an analysis, how to prove it correct, how to implement it, and how to use its results for improving the program at hand. Furthermore, the main part of the development focuses on the intuitively understandable analysis of "detection of signs" and only develops the somewhat more demanding analysis of [NN, Chapter 5] towards the end.

For a short course (based on Sections 1.2, 1.3, 4.1, 4.2, and 4.3 of [NN]) we recommend covering just Chapters 1 and 2 of this note.

Aarhus, October 1996 *H.R.Nielson & F.Nielson*

Chapter 1

Introduction

The availability of powerful tools is crucial for the design, implementation and maintenance of large programs. Advanced programming environments provide many such tools: syntax directed editors, optimizing compilers and debuggers in addition to tools for transforming programs and for estimating their performance. Program analyses play a major role in many of these tools: they are able to give useful information about the dynamic behaviour of programs without actually running them. Below we give a few examples.

In a *syntax directed editor* we may meet warnings as “variable `x` is used before it is initialised”, or “the part of the program starting at line 1298 and ending at line 1354 will never be entered”, or “there is a reference outside the bounds of array `a`”. Such information is the result of various program analyses. The first warning is the result of a *definition-use* analysis: at each point of a program where the value of a variable is used, the analysis will determine those points where the variable might have obtained its present value; if there are none then clearly the variable is uninitialised. The second warning might be the result of a *constant propagation* analysis: at each point of a program where an expression is evaluated the analysis will attempt to deduce that the expression always evaluates to a constant. So if the expression is the test of a conditional we might deduce that only one of the branches can ever be taken. The third warning could be the result of an *interval* analysis: instead of determining a possible constant value we determine upper and lower bounds of the value that the expression may evaluate to. So if the expression is an index into an array, then a comparison with the bounds of the array will suffice for issuing the third warning above.

Traditionally, program analyses have been developed for *optimizing compilers*. They are used at all levels in the compiler: some optimizations apply to the source program, others to the various intermediate representations used inside the compiler and finally there are optimizations that exploit the architecture of the target machine and therefore directly improve the target code. The improvements facilitated by these analyses, and the associated transformations, may result in dramatic reductions of the running time. One example is the *available expressions* analysis: an expression E is available at a program point p if E has been computed previously and the values of the free variables of E have not changed since then. Clearly we can avoid recomputing the expression and instead use the previously computed value. This information is particularly useful at the intermediate level in a compiler for computing actual addresses into data structures with arrays as a typical example. Another example is *live variable* analysis: given a variable x and a point p in the program will the value of x at p be used at a later stage; if so then x is said to be live at p and otherwise it is said to be dead. Such information is useful when deciding how to use the registers of a machine: the values of dead variables need not be stored in memory when the registers in which they reside are reused for other purposes.

Many *program transformations* are only valid when certain conditions are fulfilled. As an example it is only safe to move the computation of an expression outside a loop if its value is not affected by the computations in the remainder of the loop. Similarly we may only replace an expression with a variable if we know that the expression already has been evaluated in a context where it gave the same result as here and where its value has been assigned to the variable. Such information may be collected by a slight extension of an *available expression* analysis: an expression E is available in x at a program point p if E has been evaluated and assigned to x on all paths leading to p and if the values of x and the free variables of E have not changed since then.

One of the more difficult tasks in an advanced programming environment is to evaluate the *performance* of programs. Typically this falls outside the power of *automatic* program analyses and requires more user coorporation.

1.1 Side-stepping the Halting Problem

It is important to realize that *exact* answers to many of the program analyses we have mentioned above do involve solving the Halting Problem! As an example consider the program fragment

$$(\texttt{if}\ \cdots\ \texttt{then}\ \texttt{x} := 1\ \texttt{else}\ (S;\ \texttt{x} := 2));\ \texttt{y} := \texttt{x}$$

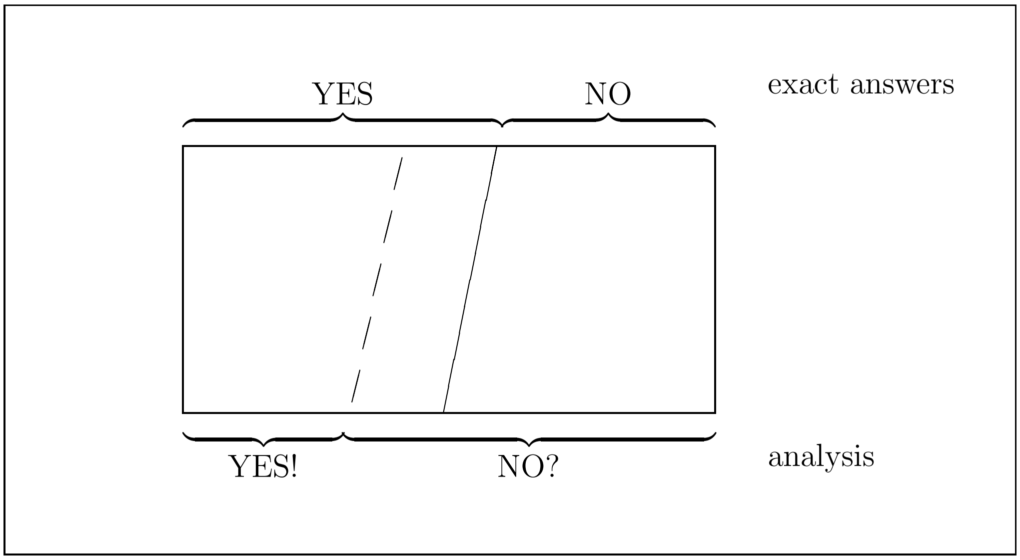


Table 1.1: Safe approximations to exact answers

and the constant propagation analysis: does `x` always evaluate to the constant 1 in the assignment to `y`? This is the case if and only if S never terminates (or if S ends with a run-time error). To allow for an implementable analysis we allow constant propagation to provide *safe approximations* to the *exact* answers: in the example we always deem that `x` does not evaluate to a constant at the assignment to `y`. In this way the possible outcomes of the analysis are

- YES!: the expression E *always* evaluates to the constant c, or
- NO?: the expression E *might not always* evaluate to a constant,

and where the second answer is *not* equivalent to saying that "the expression E *does not always* evaluate to a constant". To be useful the second answer should not be produced too often: a useless but correct analysis is obtained by always producing the second answer. We shall see that this is a general phenomenon: one answer of the analysis will imply the exact answer but the other answer will not; this is illustrated in Table 1.1 and we say that the analysis *errs on the safe side*. To be more precise about this requires a study of the concrete analysis and the use in which we intend to put it. We thus return to the issue when dealing with the correctness of a given analysis.

1.2 Classification of Program Analyses

Program analyses give information about the *dynamic* behaviour of programs. The analyses are performed *statically* meaning that the programs are *not* run on all possible inputs in order to find the result of the analysis. On the other hand the analyses are *safe* meaning that the result of the analysis describes all possible runs of the program. This effect is obtained by letting the analysis compute with *abstract properties* of the "real" values rather than with the "real" values themselves.

Program analyses are often classified in two respects:

- *what* kind of properties do they compute, and
- *how* do they compute with them.

Basically, there are two kinds of properties:

- properties of *values*, and
- properties of *relationships* between values.

The first class of analyses is often called *first order analyses* as they compute with direct properties of the values. The second class of analyses is called *second order analyses* as they compute with properties derived from relationships between values.

An example of an analysis falling within the first category is a *detection of signs* analysis. The properties of interest are the signs of values so rather than computing with numbers we will compute with the sign properties POS, ZERO, NEG and ANY. Also constant propagation and interval analysis are examples of first order analyses.

An example of an analysis falling within the second category is live variable analysis. Here the properties associated with the variables are LIVE and DEAD and obviously this does not say much about the "real" value of the variables. However, it expresses a property of the relationship between "real" values: if the property is DEAD then the variable could have any value whatsoever – the result of the computation would be the same since the value will never be used. On the other hand, if the property is LIVE then the "real" value of the variable might influence the final outcome of the computation. Detection of common subexpressions and available subexpression analysis are other examples of second order analyses.

The other classification is concerned with *how* the analyses compute with the properties. Again there are basically two approaches (although mixtures exist):

- forward analyses, and
- backwards analyses.

In a *forward analysis* the computation proceeds much as in the direct style denotational semantics: given the properties of the input the analysis will compute properties of the output. For example, the detection of signs analysis proceeds in this way.

As the name suggests, a *backwards analysis* performs the computation the other way round: given properties of the output of the computation it will predict the properties that the input should have. Here live variable analysis is a classical example: the output certainly should have the property LIVE and the idea is then to calculate backwards to see which parts of the input (and which parts of the intermediate results) really are needed in order to compute the output.

The two classifications can be freely mixed so for example the detection of signs analysis is a first order forward analysis and live variable analysis is a second order backwards analysis. An example of a first order backward analysis is an *error detection analysis*: we might want to ensure that the program does not result in an error (say, from an attempt to divide by zero) and the analysis could then provide information about inputs that definitely would prevent this from happening. This kind of information might be crucial for the design of safety critical systems. An example of a second order forward analysis is a *functional dependency analysis*: does the output of the computation depend functionally on the input or is it effected by uninitialised variables. Such information might be important during debugging to nail down spurious errors.

1.3 Ccpo's and Complete Lattices

The analyses will all work with a set $\mathbf{P}$ of *properties*. As an example, in the detection of signs analysis, $\mathbf{P}$ will contain the properties POS, ZERO, NEG and ANY (and others) and in a live variable analysis it will contain the properties LIVE and DEAD. Since some properties are more informative than others we shall equip $\mathbf{P}$ with a *partial ordering* $\sqsubseteq_P$. So for example in the detection of signs analysis we have POS $\sqsubseteq_P$ ANY because it is more informative to know that a number is positive than it is to know that it can have any sign. Similarly, in the live variable analysis we may take DEAD $\sqsubseteq_P$ LIVE because it is more informative to know that the value of a variable definitely is not used in the rest of the computation than it is to know that it might be used. (One might rightly feel that this intuition is somewhat at odds with the view point of denotational semantics; nonetheless the approach makes sense!) When specifying an analysis we shall always make sure that

$$(\mathbf{P}, \sqsubseteq_P) \text{ is a complete lattice}$$

as defined in Chapter 4 of [NN][1]. The lattice structure gives us a convenient method for *combining* properties: if some value has the two properties p_1 and p_2 then we can combine this to say that it has the property $\bigsqcup_P\{p_1, p_2\}$ where $\bigsqcup_P$ is the least upper bound operation on $\mathbf{P}$. It is convenient to write $p_1 \sqcup_P p_2$ for $\bigsqcup_P\{p_1, p_2\}$.

Many analyses (for example the detection of signs analysis) associate properties with the individual variables of the program and here the function spaces are often used to model *property states*: in the standard semantics states map variables to values whereas in the analysis the property states will map variables to their properties:

$$\mathbf{PState} = \mathbf{Var} \to \mathbf{P}$$

The property states inherit the ordering from the properties in a point-wise manner. This is in fact a corollary of a fairly general result which can be expressed as follows:

Lemma 1.1 Assume that S is a non-empty set and that $(D, \sqsubseteq)$ is a partially ordered set. Let $\sqsubseteq'$ be the ordering on the set $S \to D$ defined by

$$f_1 \ \sqsubseteq' \ f_2 \text{ if and only if } f_1\ x \ \sqsubseteq \ f_2\ x \text{ for all } x \in S$$

Then $(S \to D, \sqsubseteq')$ is a partially ordered set. Furthermore, $(S \to D, \sqsubseteq')$ is a ccpo if D is and it is a complete lattice if D is. In both cases we have

$$(\textstyle\bigsqcup' Y)\ x = \bigsqcup\{f\ x \mid f \in Y\}$$

so that least upper bounds are determined pointwise.

---

[1] [NN]: H.R.Nielson, F.Nielson: Semantics with Applications – A Formal Introduction, Wiley 1992.

Proof It is straightforward to verify that $\sqsubseteq'$ is a partial order so we omit the details. We shall first prove the lemma in the case where D is a complete lattice so let Y be a subset of $S \to D$. Then the formula

$$(\bigsqcup' Y)\ x = \bigsqcup\{f\ x \mid f \in Y\}$$

defines an element $\bigsqcup' Y$ of $S \to D$ because D being a complete lattice means that $\bigsqcup\{f\ x \mid f \in Y\}$ exists for all x of S. This shows that $\bigsqcup' Y$ is a *well-defined* element of $S \to D$. To see that $\bigsqcup' Y$ is an *upper bound* of Y let $f_0 \in Y$ and we shall show that $f_0 \sqsubseteq' \bigsqcup' Y$. This amounts to considering an arbitrary x in S and showing

$$f_0\ x \sqsubseteq \bigsqcup\{f\ x \mid f \in Y\}$$

and this is immediate because $\bigsqcup$ is the least upper bound operation in D. To see that $\bigsqcup' Y$ is the *least* upper bound of Y let f_1 be an upper bound of Y and we shall show that $\bigsqcup' Y \sqsubseteq' f_1$. This amounts to showing

$$\bigsqcup\{f\ x \mid f \in Y\} \sqsubseteq f_1\ x$$

for an arbitrary $x \in S$. However, this is immediate because $f_1\ x$ must be an upper bound of $\{f\ x \mid f \in Y\}$ and because $\bigsqcup$ is the least upper bound operation in D.

To prove the other part of the lemma assume that D is a ccpo and that Y is a chain in $S \to D$. The formula

$$(\bigsqcup' Y)\ x = \bigsqcup\{f\ x \mid f \in Y\}$$

defines an element $\bigsqcup' Y$ of $S \to D$: each $\{f\ x \mid f \in Y\}$ will be a chain in D because Y is a chain and hence each $\bigsqcup\{f\ x \mid f \in Y\}$ exists because D is a ccpo. That $\bigsqcup' Y$ is the least upper bound of Y in $S \to D$ follows as above. $\square$

Chapter 2

Detection of Signs Analysis

In this chapter we restrict our attention to first order forward analyses. Thus we shall be interested in properties of the values computed by the standard semantics and the analysis will proceed in a forward manner just as the standard semantics. We shall focus on one particular analysis, the detection of signs analysis, and leave the development of other analyses to the exercises. First we show how to *specify* the analysis and next we prove its *existence*. We then show that it is a *safe* approximation of the standard semantics. Finally, we address the *applicability* of the analysis in program transformations.

2.1 Detection of Signs Analysis

The rules for computation with signs are well-known and the idea is now to turn them into an analysis of programs in the **While** language. The specification of the analysis falls into two parts. First we have to specify the properties that we compute with: in this case properties of numbers and truth values. Next we specify the analysis itself for the three syntactic categories: arithmetic expressions, boolean expressions, and statements.

The detection of signs analysis is based on three basic properties of numbers:

- POS: the number is positive,
- ZERO: the number is zero, and
- NEG: the number is negative.

Although a given number will have one (and only one) of these properties it is obvious that we easily loose precision when calculating with signs: the subtraction of two positive numbers may give any number so the sign of the result cannot be described by one of the three basic properties. This is a common situation in program analysis and the solution is to introduce extra properties that express *combinations* of the basic properties. For the detection of signs analysis we may add the following properties:

- NON-NEG: the number is not negative,
- NON-ZERO: the number is not zero,
- NON-POS: the number is not positive, and
- ANY: the number can have any sign.

For each property we can determine a set of numbers that are described by that property. When formulating the analysis it is convenient to have a property corresponding to the *empty set* of numbers as well and we therefore introduce the property

- NONE: the number belongs to $\emptyset$.

Let now **Sign** be the set

$$\{\textsc{neg}, \textsc{zero}, \textsc{pos}, \textsc{non-pos}, \textsc{non-zero}, \textsc{non-neg}, \textsc{any}, \textsc{none}\}$$

We shall equip **Sign** with a partial ordering $\sqsubseteq_S$ reflecting the subset ordering on the underlying sets of numbers. The ordering is depicted by means of the Hasse diagram of Table 2.1. So for example POS $\sqsubseteq_S$ NON-ZERO holds because $\{z \in \mathbf{Z} \mid z > 0\} \subseteq \{z \in \mathbf{Z} \mid z \neq 0\}$ and NONE $\sqsubseteq_S$ NEG holds because $\emptyset \subseteq \{z \in \mathbf{Z} \mid z < 0\}$.

Exercise 2.1 Show that (**Sign**, $\sqsubseteq_S$) is a complete lattice and let $\bigsqcup_S$ be the associated least upper bound operation. For each pair p_1 and p_2 of elements from **Sign** specify $p_1 \sqcup_S p_2$. □

Clearly we can associate a "best" property with each number. To formalise this we define a function

$$\mathsf{abs}_Z\colon \mathbf{Z} \to \mathbf{Sign}$$

that will abstract a number into its sign:

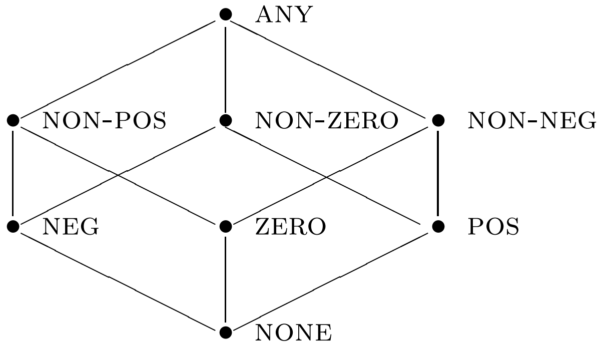


Table 2.1: (**Sign**, $\sqsubseteq_S$)

$$\mathsf{abs}_Z\ z = \begin{cases} \text{NEG} & \text{if } z < 0 \\ \text{ZERO} & \text{if } z = 0 \\ \text{POS} & \text{if } z > 0 \end{cases}$$

In the detection of signs analysis we define the set **PState** of property states by

$$\mathbf{PState} = \mathbf{Var} \to \mathbf{Sign}$$

and we shall use the meta-variable ps to range over **PState**. The operation abs_Z is lifted to states

$$\mathsf{abs}\colon \mathbf{State} \to \mathbf{PState}$$

by defining it in a component wise manner: $(\mathsf{abs}\ s)\ x = \mathsf{abs}_Z\ (s\ x)$ for all variables x.

From Lemma 1.1 we have

Corollary 2.2 Let $\sqsubseteq_{PS}$ be the ordering on **PState** defined by

$$ps_1 \ \sqsubseteq_{PS}\ ps_2 \text{ if and only if } ps_1\ x\ \sqsubseteq_S\ ps_2\ x \text{ for all } x \in \mathbf{Var}.$$

Then $(\mathbf{PState}, \sqsubseteq_{PS})$ is a complete lattice. In particular the least upper bound $\bigsqcup_{PS} Y$ of a subset Y of **PState** is characterized by

$$(\textstyle\bigsqcup_{PS} Y)\ x = \bigsqcup_S \{ps\ x \mid ps \in Y\}$$

and is thus defined in a componentwise manner.

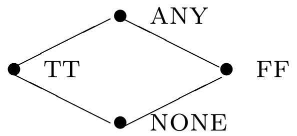


Table 2.2: $(\mathbf{TT}, \sqsubseteq_T)$

We shall write INIT for the *least element* of **PState**, that is for the property state that maps all variables to NONE.

In the analysis we compute with the properties of **Sign** rather than the numbers of **Z**. Similarly, we will have to replace the truth values **T** with some set of properties: although knowledge about the signs may enable us to predict the truth value of certain boolean expressions this need not always be the case. We shall therefore introduce four properties corresponding to the four subsets of the truth values:

- TT: corresponding to the set $\{\mathbf{tt}\}$,
- FF: corresponding to the set $\{\mathbf{ff}\}$,
- ANY: corresponding to the set $\{\mathbf{tt}, \mathbf{ff}\}$, and
- NONE: corresponding to the set $\emptyset$.

The set is equiped with an ordering $\sqsubseteq_T$ reflecting the subset ordering on the sets of truth values. This is depicted in Table 2.2.

Exercise 2.3 Show that $(\mathbf{TT}, \sqsubseteq_T)$ is a complete lattice and let $\bigsqcup_T$ be the associated least upper bound operation. For each pair p_1 and p_2 of elements from **TT** specify $p_1 \sqcup_T p_2$. $\square$

We shall also introduce an abstraction function for truth values. It has the functionality

$$\mathsf{abs}_T\colon \mathbf{T} \to \mathbf{TT}$$

and is defined by $\mathsf{abs}_T\ \mathbf{tt} = $ TT and $\mathsf{abs}_T\ \mathbf{ff} = $ FF.

Analysis of expressions

Recall that the semantics of arithmetic expression is given by a function

$$
\begin{array}{lcl}
\mathcal{SA}[\![n]\!]ps & = & \mathsf{abs}_Z(\mathcal{N}[\![n]\!]) \\
\mathcal{SA}[\![x]\!]ps & = & ps\ x \\
\mathcal{SA}[\![a_1 + a_2]\!]ps & = & \mathcal{SA}[\![a_1]\!]ps\ +_S\ \mathcal{SA}[\![a_2]\!]ps \\
\mathcal{SA}[\![a_1\ *\ a_2]\!]ps & = & \mathcal{SA}[\![a_1]\!]ps\ *_S\ \mathcal{SA}[\![a_2]\!]ps \\
\mathcal{SA}[\![a_1 - a_2]\!]ps & = & \mathcal{SA}[\![a_1]\!]ps\ -_S\ \mathcal{SA}[\![a_2]\!]ps
\end{array}
$$

Table 2.3: Detection of signs analysis of arithmetic expressions

$$\mathcal{A}\text{: } \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Z}$$

In the analysis we do not know the exact value of the variables but only their properties and consequently we can only compute a property of the arithmetic expression. So the analysis will be given by a function

$$\mathcal{SA}\text{: } \mathbf{Aexp} \to \mathbf{PState} \to \mathbf{Sign}$$

whose defining clauses are given in Table 2.3.

In the clause for n we use the function abs_Z to determine the property of the corresponding number. For variables we simply consult the property state. For the composite constructs we proceed in a compositional manner and rely on addition, multiplication and subtraction operations defined on **Sign**. For addition the operation $+_S$ is written in detail in Table 2.4. The multiplication and subtraction operations $*_S$ and $-_S$ are only partly specified in that table.

The semantics of boolean expressions is given by a function

$$\mathcal{B} : \mathbf{Bexp} \ \to\ \mathbf{State}\ \to\ \mathbf{T}$$

As in the case of arithmetic expressions the analysis will use property states rather than states. The truth values will be replaced by the set **TT** of properties of truth values so the analysis will be given by a function

$$\mathcal{SB} : \mathbf{Bexp} \ \to\ \mathbf{PState}\ \to\ \mathbf{TT}$$

whose defining clauses are given in Table 2.5.

The clauses for `true` and `false` should be straightforward. For the tests on arithmetic expressions we rely on operations defined on the lattice **Sign** and giving results in **TT**; these operations are partly specified in Table 2.6. In the case of negation and conjunction we need similar operations defined on **TT** and these are also specified in Table 2.6.

$+_S$	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE
NEG	NONE	NEG	NEG	ANY	NEG	ANY	ANY	ANY
ZERO	NONE	POS	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
POS	NONE	ANY	POS	POS	ANY	ANY	POS	ANY
NON-POS	NONE	NEG	NON-POS	ANY	NON-POS	ANY	ANY	ANY
NON-ZERO	NONE	ANY	NON-ZERO	ANY	ANY	ANY	ANY	ANY
NON-NEG	NONE	ANY	NON-NEG	POS	ANY	ANY	NON-NEG	ANY
ANY	NONE	ANY	ANY	ANY	ANY	ANY	ANY	ANY

$*_S$	NEG	ZERO	POS
NEG	POS	ZERO	NEG
ZERO	ZERO	ZERO	ZERO
POS	NEG	ZERO	POS

$-_S$	NEG	ZERO	POS
NEG	ANY	NEG	NEG
ZERO	POS	ZERO	NEG
POS	POS	POS	ANY

Table 2.4: Operations on **Sign**

$$\begin{aligned}
\mathcal{SB}[\![\texttt{true}]\!]ps &= \text{TT}\\
\mathcal{SB}[\![\texttt{false}]\!]ps &= \text{FF}\\
\mathcal{SB}[\![a_1 = a_2]\!]ps &= \mathcal{SA}[\![a_1]\!]ps \ =_S\ \mathcal{SA}[\![a_2]\!]ps\\
\mathcal{SB}[\![a_1 \leq a_2]\!]ps &= \mathcal{SA}[\![a_1]\!]ps \ \leq_S\ \mathcal{SA}[\![a_2]\!]ps\\
\mathcal{SB}[\![\neg b]\!]ps &= \neg_T\ (\mathcal{SB}[\![b]\!]ps)\\
\mathcal{SB}[\![b_1 \wedge b_2]\!]ps &= \mathcal{SB}[\![b_1]\!]ps \ \wedge_T\ \mathcal{SB}[\![b_2]\!]ps
\end{aligned}$$

Table 2.5: Detection of signs analysis of boolean expressions

Example 2.4 We have $\mathcal{SB}[\![\neg\texttt{(x=1)}]\!]ps = \neg_T(ps\ \texttt{x} =_S \text{POS})$ which can be represented by the following table

ps `x`	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
$\mathcal{SB}[\![\neg\texttt{(x=1)}]\!]ps$	NONE	TT	TT	ANY	TT	ANY	ANY	ANY

Thus if `x` is positive we cannot deduce anything about the outcome of the test whereas if `x` is negative then the test will always give true. □

$=_S$	NEG	ZERO	POS
NEG	ANY	FF	FF
ZERO	FF	TT	FF
POS	FF	FF	ANY

$\leq_S$	NEG	ZERO	POS
NEG	ANY	TT	TT
ZERO	FF	TT	TT
POS	FF	FF	ANY

$\neg_T$	
NONE	NONE
TT	FF
FF	TT
ANY	ANY

$\wedge_T$	NONE	TT	FF	ANY
NONE	NONE	NONE	NONE	NONE
TT	NONE	TT	FF	ANY
FF	NONE	FF	FF	FF
ANY	NONE	ANY	FF	ANY

Table 2.6: Operations on **Sign** and **TT**

Analysis of statements

In the denotational semantics the meaning of statements is given by a function

$$\mathcal{S}_{ds}\text{: } \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State}).$$

In the analysis we perform two changes: First, we replace the states by property states so that given information about the signs of the variables *before* the statement is executed we will obtain information about the signs of the variables *after* the execution has (possibly) terminated. Second, we replace partial functions by *total* functions; this is a crucial change in that the whole point of performing a program analysis is to trade precision for termination. So the analysis will be specified by a function

$$\mathcal{SS}\text{: } \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

whose clauses are given in Table 2.7.

At the surface these clauses are exactly as those of the direct style denotational semantics in Chapter 4 of [NN]. However, the auxiliary function cond_S is different in that it has to take into account that the outcome of the test can be any of the *four* properties of **TT**. We shall define it by

$$\begin{aligned}&\mathsf{cond}_S(f, h_1, h_2)ps\\ &\qquad = \begin{cases} \text{INIT} & \text{if } f\ ps = \text{NONE}\\ h_1\ ps & \text{if } f\ ps = \text{TT}\\ h_2\ ps & \text{if } f\ ps = \text{FF}\\ (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \text{ANY}\end{cases}\end{aligned}$$

$$\mathcal{SS}[\![x := a]\!]ps = ps[x \mapsto \mathcal{SA}[\![a]\!]ps]$$

$$\mathcal{SS}[\![\texttt{skip}]\!] = \mathsf{id}$$

$$\mathcal{SS}[\![S_1; S_2]\!] = \mathcal{SS}[\![S_2]\!] \circ \mathcal{SS}[\![S_1]\!]$$

$$\mathcal{SS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] = \mathsf{cond}_S(\mathcal{SB}[\![b]\!], \mathcal{SS}[\![S_1]\!], \mathcal{SS}[\![S_2]\!])$$

$$\mathcal{SS}[\![\texttt{while } b \texttt{ do } S]\!] = \mathsf{FIX}\ H \quad \text{where } H\ h = \mathsf{cond}_S(\mathcal{SB}[\![b]\!], h \circ \mathcal{SS}[\![S]\!], \mathsf{id})$$

Table 2.7: Detection of signs analysis of statements

Here the operation $\sqcup_{PS}$ is the *binary* least upper bound operation of **PState**:

$$ps_1 \sqcup_{PS} ps_2 = \bigsqcup_{PS}\{ps_1, ps_2\}$$

Since (**PState**, $\sqsubseteq_{PS}$) is a complete lattice (Corollary 2.2) this is a well-defined operation. The idea behind the definition of cond_S is that if the outcome of the test is TT or FF then we know exactly which branch will be taken when executing the statement. So we select the analysis of that branch to be the result of analysing the conditional. If the outcome of the test is ANY then the execution of the conditional might result in **tt** as well as **ff** so in the analysis we have to combine the results of the two branches. So if one branch says that `x` has the property POS and the other says that it has the property NEG then we can only deduce that after execution of the conditional `x` will be either positive or negative and this is exactly what is achieved by using the least upper bound operation. The case where the test gives NONE should not be possible so in this case we return the property state INIT that does not describe any states.

In the clause for the `while`-loop we also use the function cond_S and otherwise the clause is as in the direct style denotational semantics. In particular we use the fixed point operation FIX as it corresponds to unfolding the `while`-loop a number of times — once for each time the *analysis* traverses the loop. As in Chapter 4 of [NN] the fixed point is defined by

$$\mathsf{FIX}\ H = \bigsqcup\{H^n\bot \mid n \geq 0\}$$

where the functionality of H is

$$H\colon (\mathbf{PState} \to \mathbf{PState}) \to (\mathbf{PState} \to \mathbf{PState})$$

and where **PState** $\rightarrow$ **PState** is the set of total functions from **PState** to **PState**. In order for this to make sense H must be a continuous function on a ccpo with least element $\perp$. We shall shortly verify that this is indeed the case.

Example 2.5 We are now in a position where we can attempt the application of the analysis to the factorial statement:

$$\mathcal{SS}[\![\texttt{y := 1; while } \neg\texttt{(x = 1) do (y := y} \star \texttt{x; x := x} - \texttt{1)}]\!]$$

We shall apply this function to the property state ps_0 that maps `x` to POS and all other variables (including `y`) to ANY. So we are interested in

$$(\mathsf{FIX}\ H)\ (ps_0[\texttt{y} \mapsto \textsc{pos}])$$

where

$$H\ h = \mathsf{cond}_S(\mathcal{SB}[\![\neg\texttt{(x = 1)}]\!], h \circ h_{fac}, \mathsf{id})$$
$$h_{fac} = \mathcal{SS}[\![\texttt{y := y} \star \texttt{x; x := x} - \texttt{1}]\!]$$

Thus we have to compute the approximations $H^0\ \perp$, $H^1\ \perp$, $H^2\ \perp \cdots$. Below we shall show that

$$H^3\ \perp\ (ps_0[\texttt{y} \mapsto \textsc{pos}])\ \texttt{y} = \textsc{any}.$$

Since $H^3\ \perp\ \sqsubseteq \mathsf{FIX}\ H$ we have

$$H^3\ \perp\ (ps_0[\texttt{y} \mapsto \textsc{pos}]) \sqsubseteq_{PS} (\mathsf{FIX}\ H)\ (ps_0[\texttt{y} \mapsto \textsc{pos}])$$

and thereby

$$H^3\ \perp\ (ps_0[\texttt{y} \mapsto \textsc{pos}])\ \texttt{y} \sqsubseteq_{P} (\mathsf{FIX}\ H)\ (ps_0[\texttt{y} \mapsto \textsc{pos}])\ \texttt{y}.$$

Thus $(\mathsf{FIX}\ H)\ (ps_0[\texttt{y} \mapsto \textsc{pos}])\ \texttt{y} = \textsc{any}$ must be the case. Thus even though we start by the assumption that `x` is positive the analysis cannot deduce that the factorial of `x` is positive. We shall remedy this shortly.

Using the definition of h_{fac} and $\mathcal{SB}[\![\neg\texttt{(x = 1)}]\!]$ (as tabulated in Example 2.4) we get

$$\begin{aligned}
&H^3 \perp (ps_0[\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = H(H^2 \perp)\ (ps_0[\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = (H^2 \perp)\ (h_{fac}(ps_0[\mathsf{y}{\mapsto}\text{POS}])) \\
&\qquad\quad \sqcup_{PS}\ \mathsf{id}\ (ps_0[\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = H(H^1 \perp)\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{POS}]) \\
&\qquad\quad \sqcup_{PS}\ (ps_0[\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = (H^1 \perp)\ (h_{fac}\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{POS}])) \\
&\qquad\quad \sqcup_{PS}\ \mathsf{id}\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{POS}]) \sqcup_{PS} (ps_0[\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = H(H^0 \perp)\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{ANY}]) \\
&\qquad\quad \sqcup_{PS}\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = (H^0 \perp)\ (h_{fac}\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{ANY}])) \\
&\qquad\quad \sqcup_{PS}\ \mathsf{id}(ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{ANY}]) \sqcup_{PS} (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{POS}]) \\
&\quad = \text{INIT} \sqcup_{PS}\ (ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{ANY}]) \\
&\quad = ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{ANY}]
\end{aligned}$$

Thus we see that the 'mistake' was made when we applied h_{fac} to the property state $ps_0[\mathsf{x}{\mapsto}\text{ANY}][\mathsf{y}{\mapsto}\text{POS}])$. □

Remark A more precise analysis may be performed if we change the definition of cond_S in the case where $f\ ps = \text{ANY}$. To do so we first introduce the notion of an *atomic* property state: ps is atomic if there exists a state s such that $\mathsf{abs}(s) = ps$. Equivalently, ps is atomic if

$$\forall x \in \mathbf{Var}\colon\ ps\ x \in \{\text{NEG, ZERO, POS}\}$$

Clearly, not all property states are atomic but all property states can be described as the least upper bound of a set of atomic property states:

$$ps = \bigsqcup_{PS} \{ps' \mid ps' \sqsubseteq_{PS} ps, \text{ and } ps' \text{ is atomic}\}$$

holds for all property states ps. If $f\ ps' = \text{TT}$ then we know that only the true branch may be taken, if $f\ ps' = \text{FF}$ then only the false branch may be taken and only in the case where $f\ ps' = \text{ANY}$ we need to take both possibilities into account. So we define the two sets $\mathsf{filter}_T(f, ps)$ and $\mathsf{filter}_F(f, ps)$ by

$$\begin{aligned}
\mathsf{filter}_T(f, ps) &= \{ps' \mid ps' \sqsubseteq_{PS} ps,\ ps' \text{ is atomic},\ \text{TT} \sqsubseteq_T f\ ps'\} \\
\mathsf{filter}_F(f, ps) &= \{ps' \mid ps' \sqsubseteq_{PS} ps,\ ps' \text{ is atomic},\ \text{FF} \sqsubseteq_T f\ ps'\}
\end{aligned}$$

We can now replace $(h_1\ ps) \sqcup_{PS} (h_2\ ps)$ in the definition of cond_S by

$$(h_1\ (\bigsqcup_{PS}\ \mathsf{filter}_T(f, ps))) \sqcup_{PS} (h_2\ (\bigsqcup_{PS}\ \mathsf{filter}_F(f, ps)))$$

Here $\bigsqcup_{PS}$ is used to combine the set of property states into a single property state before applying the analysis of the particular branch to it.

We note in passing that an even more precise analysis might result if the use of $\bigsqcup_{PS}$ was postponed until after h_i had been applied pointwise to the property states in the corresponding set. □

Example 2.6 To be able to obtain useful information from the analysis of the factorial statement we need to do two things:

- use the definition of cond_S given in the previous remark, and
- rewrite the test of the factorial statement to use $\neg(\mathtt{x}\leq\mathtt{1})$ instead of $\neg(\mathtt{x=1})$.

With these amendments we are interested in

$$\mathcal{SS}[\![\mathtt{y\ :=\ 1;\ while\ }\neg\mathtt{(x\ }\leq\mathtt{\ 1)\ do\ (y\ :=\ y\ *\ x;\ x\ :=\ x\ -\ 1)}]\!]\, ps_0$$

where $ps_0\ \mathtt{x} = \textsc{pos}$ and ps_0 maps all other variables to $\textsc{any}$. So we are interested in

$$(\mathsf{FIX}\ H)\ (ps_0[\mathrm{y} \mapsto \textsc{pos}])$$

where

$$H\ h = \mathsf{cond}_S(\mathcal{SB}[\![\neg(\mathtt{x} \leq \mathtt{1})]\!], h\ \circ\ h_{fac}, \mathsf{id})$$
$$h_{fac} = \mathcal{SS}[\![\mathtt{y\ :=\ y\ *\ x;\ x\ :=\ x\ -\ 1}]\!]$$

In the case where $ps\ \mathtt{x} = p \in \{\textsc{pos}, \textsc{any}\}$ and $ps\ \mathtt{y} = \textsc{pos}$ we have

$$\begin{aligned} H\ h\ ps &= (h \circ h_{fac})(\textstyle\bigsqcup_{PS} \mathsf{filter}_T(\mathcal{SB}[\![\neg(\mathtt{x} \leq \mathtt{1})]\!], ps)) \\ &\qquad \sqcup_{PS}\ \mathsf{id}\ (\textstyle\bigsqcup_{PS} \mathsf{filter}_F(\mathcal{SB}[\![\neg(\mathtt{x} \leq \mathtt{1})]\!], ps)) \\ &= (h \circ h_{fac})(ps[\mathtt{x} \mapsto \textsc{pos}]) \\ &\qquad \sqcup_{PS}\ (ps[\mathtt{x} \mapsto p]) \\ &= h(ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} ps \end{aligned}$$

We can now compute the iterands $H^n \perp ps$ as follows when $ps\ \mathtt{x} = p \in \{\textsc{pos}, \textsc{any}\}$ and $ps\ \mathtt{y} = \textsc{pos}$:

$$\begin{array}{rcl} H^0 \perp ps & = & \textsc{init} \\ H^1 \perp ps & = & H^0 \perp (ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} (ps[\mathtt{x} \mapsto p]) \\ & = & ps \\ H^2 \perp ps & = & H^1 \perp (ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} (ps[\mathtt{x} \mapsto p]) \\ & = & ps[\mathtt{x} \mapsto \textsc{any}] \\ H^3 \perp ps & = & H^2 \perp (ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} (ps[\mathtt{x} \mapsto p]) \\ & = & ps[\mathtt{x} \mapsto \textsc{any}] \end{array}$$

One can show that for all $n \geq 2$

$$H^n \perp ps = ps[\mathtt{x} \mapsto \textsc{any}]$$

when $ps\ \mathtt{x} \in \{\textsc{pos}, \textsc{any}\}$ and $ps\ \mathtt{y} = \textsc{pos}$. It then follows that

$$(\mathsf{FIX}\ H)(ps_0[\mathtt{y} \mapsto \textsc{pos}]) = ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}]$$

So the analysis tells us that if x is positive in the initial state then y will be positive in the final state (provided that the program terminates). □

Exercise 2.7 Show that if H: $(\mathbf{PState} \to \mathbf{PState}) \to (\mathbf{PState} \to \mathbf{PState})$ is continuous (or just monotone) and

$$H^n \perp = H^{n+1} \perp$$

for some n then $H^{n+m} \perp = H^{n+1+m} \perp$ for all $m > 0$. Conclude that

$$H^n \perp = H^m \perp$$

for $m \geq n$ and therefore $\mathsf{FIX}\ H = H^n \perp$.

Show by means of an example that $H^n \perp ps_0 = H^{n+1} \perp ps_0$ for some $ps_0 \in \mathbf{PState}$ does not necessarily imply that $\mathsf{FIX}\ H\ ps_0 = H^n \perp ps_0$. □

Exercise 2.8 A constant propagation analysis attempts to predict whether arithmetic and booleans expressions always evaluate to constant values. For natural numbers the following properties are of interest:

- ANY: it can be any number,

- z: it is definitely the number $z \in \mathbf{Z}$, and

- NONE: the number belongs to $\emptyset$

Let now $\mathbf{Const} = \mathbf{Z} \cup \{\text{ANY}, \text{NONE}\}$ and let $\sqsubseteq_C$ be the ordering defined by

- $\text{NONE} \sqsubseteq_C p$ for all $p \in \mathbf{Const}$, and

- $p \sqsubseteq_C \text{ANY}$ for all $p \in \mathbf{Const}$

All other elements are incomparable. Draw the Hasse-diagram for $(\mathbf{Const}, \sqsubseteq_C)$.

Let $\mathbf{PState} = \mathbf{Var} \to \mathbf{Const}$ be the set of property states and let $\mathbf{TT}$ be the properties of the truth values. Specify the constant propagation analysis by defining the functionals

$$\mathcal{CA}\colon \mathbf{Aexp} \to \mathbf{PState} \to \mathbf{Const}$$
$$\mathcal{CB}\colon \mathbf{Bexp} \to \mathbf{PState} \to \mathbf{TT}$$
$$\mathcal{CS}\colon \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

Be careful to specify the auxiliary operations in detail. Give a couple of examples illustrating the power of the analysis. □

2.2 Existence of the Analysis

Having specified the detection of signs analysis we shall now show that it is indeed well-defined. We proceed in three stages:

- First we introduce a partial order on $\mathbf{PState} \to \mathbf{PState}$ such that it becomes a ccpo.

- Then we show that certain auxiliary functions used in the definition of $\mathcal{SS}$ are continuous.

- Finally we show that the fixed point operator only is applied to continuous functions.

The ccpo

Thus our first task is to define a partial order on **PState** $\rightarrow$ **PState** and for this we use the approach developed in Lemma 1.1. Instantiating the non-empty set S to the set **PState** and the partially ordered set $(D, \sqsubseteq)$ to $(\mathbf{PState}, \sqsubseteq_{DS})$ we get:

Corollary 2.9 Let $\sqsubseteq$ be the ordering on **PState** $\rightarrow$ **PState** defined by

$$h_1 \sqsubseteq h_2 \text{ if and only if } h_1\ ps \sqsubseteq_{PS} h_2\ ps \text{ for all } ps \in \mathbf{PState}.$$

Then $(\mathbf{PState} \rightarrow \mathbf{PState}, \sqsubseteq)$ is a complete lattice, and hence a ccpo, and the formula for least upper bounds is

$$(\bigsqcup Y)\ ps = \bigsqcup_{PS} \{h\ ps \mid h \in Y\}$$

for all subsets Y of **PState** $\rightarrow$ **PState**.

Auxiliary functions

Our second task is to ensure that the function H used in Table 2.7 is a continuous function from **PState** $\rightarrow$ **PState** to **PState** $\rightarrow$ **PState**. For this we follow the approach of Section 4.3 and show that cond_S is continuous in its second argument and later that composition is continuous in its first argument.

Lemma 2.10 Let f: **PState** $\rightarrow$ **TT**, h_0: **PState** $\rightarrow$ **PState** and define

$$H\ h = \mathsf{cond}_S(f,\ h,\ h_0)$$

Then H: $(\mathbf{PState} \rightarrow \mathbf{PState}) \rightarrow (\mathbf{PState} \rightarrow \mathbf{PState})$ is a continuous function.

Proof We shall first prove that H is *monotone* so let h_1 and h_2 be such that $h_1 \sqsubseteq h_2$, that is $h_1\ ps \sqsubseteq_{PS} h_2\ ps$ for all property states ps. We then have to show that $\mathsf{cond}_S(f,\ h_1,\ h_0)\ ps \sqsubseteq_{PS} \mathsf{cond}_S(f,\ h_2,\ h_0)\ ps$ for all property states ps. The proof is by cases on the value of $f\ ps$. If $f\ ps$ = NONE then the result trivially holds. If $f\ ps$ = TT then the result follows from the assumption

$$h_1\ ps\ \sqsubseteq_{PS}\ h_2\ ps.$$

If $f\ ps = \text{FF}$ then the result trivially holds. If $f\ ps = \text{ANY}$ then the result follows from

$$(h_1\ ps \sqcup_{PS} h_0\ ps) \sqsubseteq_{PS}\ (h_2\ ps \sqcup_{PS} h_0\ ps)$$

which again follows from the assumption $h_1\ ps\ \sqsubseteq_{PS}\ h_2\ ps$.

To see that H is *continuous* let Y be a non-empty chain in **PState** $\rightarrow$ **PState**. Using the characterization of least upper bounds in **PState** given in Corollary 2.9 we see that we must show that

$$(H(\bigsqcup Y))\ ps = \bigsqcup_{PS}\ \{(H\ h)ps \mid h\ \in\ Y\}$$

for all property states ps in **PState**. The proof is by cases on the value of $f\ ps$. If $f\ ps = \text{NONE}$ then we have $(H\ (\bigsqcup Y))\ ps = \text{INIT}$ and

$$\begin{aligned}\bigsqcup_{PS}\ \{(H\ h)ps \mid h\ \in\ Y\} &= \bigsqcup_{PS}\ \{\text{INIT} \mid h\ \in\ Y\}\\ &= \text{INIT}\end{aligned}$$

Thus we have proved the required result in this case. If $f\ ps = \text{TT}$ then we have

$$\begin{aligned}(H(\bigsqcup Y))ps &= (\bigsqcup Y)ps\\ &= (\bigsqcup_{PS}\{h\ ps \mid h\ \in\ Y\})\end{aligned}$$

using the characterization of least upper bounds and furthermore

$$\bigsqcup_{PS}\{(H\ h)ps \mid h\ \in\ Y\} = \bigsqcup_{PS}\{h\ ps \mid h\ \in\ Y\}$$

and the result follows. If $f\ ps = \text{FF}$ then we have

$$(H(\bigsqcup Y))ps = h_0\ ps$$

and

$$\begin{aligned}\bigsqcup_{PS}\{(H\ h)ps \mid h\ \in\ Y\} &= \bigsqcup_{PS}\{h_0\ ps \mid h\ \in\ Y\}\\ &= h_0\ ps\end{aligned}$$

where the last equality follows because Y is non-empty. If $f\ ps = \text{ANY}$ then the characterization of least upper bounds in **PState** gives:

$$\begin{aligned}(H(\bigsqcup Y))ps &= ((\bigsqcup Y)ps) \sqcup_{PS} (h_0\ ps)\\ &= (\bigsqcup_{PS}\{h\ ps \mid h \in Y\}) \sqcup_{PS} (h_0\ ps)\\ &= \bigsqcup_{PS}\{h\ ps \mid h \in Y \cup \{h_0\}\}\end{aligned}$$

and

$$\begin{aligned}\bigsqcup_{PS}\{(H\ h)ps \mid h \in Y\} &= \bigsqcup_{PS}\{(h\ ps) \sqcup_{PS} (h_0\ ps) \mid h \in Y\}\\ &= \bigsqcup_{PS}\{h\ ps \mid h \in Y \cup \{h_0\}\}\end{aligned}$$

where the last equality follows because Y is non-empty. Thus H is continuous. □

Exercise 2.11 Let f: **PState** → **TT**, h_0: **PState** → **PState** and define

$$H\ h = \mathsf{cond}_S(f,\ h_0,\ h)$$

Show that H: (**PState** → **PState**) → (**PState** → **PState**) is a continuous function. □

Lemma 2.12 Let h_0: **PState** → **PState** and define

$$H\ h = h \circ h_0$$

Then H: (**PState** → **PState**) → (**PState** → **PState**) is a continuous function.

Proof We shall first show that H is *monotone* so let h_1 and h_2 be such that $h_1 \sqsubseteq h_2$, that is $h_1\ ps \sqsubseteq_{PS} h_2\ ps$ for all property states ps. Clearly we then have $h_1(h_0\ ps) \sqsubseteq_{PS} h_2(h_0\ ps)$ for all property states ps and thereby we have proved the monotonicity of H.

To prove the *continuity* of H let Y be a non-empty chain in **PState** → **PState**. We must show that

$$(H(\bigsqcup Y))ps = (\bigsqcup\{H\ h \mid h \in Y\})ps$$

for all property states ps. Using the characterization of least upper bounds given in Corollary 2.9 we get

$$\begin{aligned}(H(\bigsqcup Y))ps &= ((\bigsqcup Y) \circ h_0)ps\\ &= (\bigsqcup Y)(h_0\ ps)\\ &= \bigsqcup_{PS}\{h(h_0\ ps) \mid h \in Y\}\end{aligned}$$

and

$$\begin{aligned}(\bigsqcup\{H\ h \mid h \in Y\})ps &= \bigsqcup_{PS}\{(H\ h)ps \mid h \in Y\}\\ &= \bigsqcup_{PS}\{(h \circ h_0)ps \mid h \in Y\}\end{aligned}$$

Hence the result follows. □

Exercise 2.13 Show that there exists h_0: **PState** → **PState** such that H defined by $H\ h = h_0 \circ h$ is *not even* a monotone function from **PState** → **PState** to **PState** → **PState** . □

Remark The example of the above exercise indicates a major departure from the secure world of Chapter 4 of [NN]. Luckily an insurance policy can be arranged. The premium is to replace all occurrences of

PState → **PState**, **PState** → **Sign**, and **PState** → **TT**

by

[**PState** → **PState**], [**PState** → **Sign**], and [**PState** → **TT**]

where $[D \to E] = \{\ f : D \to E \mid f \text{ is continuous }\}$. One can then show that $[D \to E]$ is a ccpo if D and E are and that the characterization of least upper bounds given in Lemma 1.1 still holds. Furthermore, the entire development in this section still carries through although there are additional proof obligations to be carried out. In this setting one gets that if h_0: [**PState** → **PState**] then H defined by $H\ h = h_0 \circ h$ is a continuous function from [**PState** → **PState**] to [**PState** → **PState**]. □

Well-definedness

First we note that the equations of Tables 2.3 and 2.5 define total functions $\mathcal{SA}$ and $\mathcal{SB}$ in **Aexp** → **PState** → **Sign** and **Bexp** → **PState** → **TT**, respectively. For the well-definedness of $\mathcal{SS}$ we have:

Proposition 2.14 The function $\mathcal{SS}$: **Stm** $\to$ **PState** $\to$ **PState** of Table 2.7 is a well-defined function.

Proof We prove by structural induction on S that $\mathcal{SS}[\![S]\!]$ is well-defined and only the case of the `while`-loop is interesting. We note that the function H used in Table 2.7 is given by

$$H = H_1 \circ H_2$$

where

$$H_1\ h = \mathsf{cond}_S(\mathcal{SB}[\![b]\!],\ h,\ \mathsf{id})$$
$$H_2\ h = h \circ \mathcal{SS}[\![S]\!]$$

Since H_1 and H_2 are continuous functions by Lemmas 2.10 and 2.12 we have that H is a continuous function by [NN, Lemma 4.35]. Hence $\mathsf{FIX}\ H$ is well-defined and this completes the proof. □

Exercise 2.15 Extend **While** with the statement `repeat` S `until` b and give the new (compositional) clause for $\mathcal{SS}$. Motivate your extension and validate the well-definedness. □

Exercise 2.16 Show that the constant propagation analysis specified in Exercise 2.8 is indeed well-defined. □

2.3 Safety of the Analysis

In this section we shall show that the analysis functions $\mathcal{SA}$, $\mathcal{SB}$ and $\mathcal{SS}$ are safe with respect to the semantic functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{S}_{ds}$. We begin with the rather simple case of expressions.

Expressions

Let g: **State** $\to$ **Z** be a function, perhaps of the form $\mathcal{A}[\![a]\!]$ for some arithmetic expression $a \in$ **Aexp**, and let h: **PState** $\to$ **Sign** be another function, perhaps of the form $\mathcal{SA}[\![a]\!]$ for some arithmetic expression $a \in$ **Aexp**. We shall introduce a relation

$$g \ \mathsf{safe}_A \ h$$

for expressing when the analysis h is *safe with respect to* the semantics g. It is defined by

$$\begin{array}{l} \mathsf{abs}(s) \sqsubseteq_{PS} \ ps \\ \Downarrow \\ \mathsf{abs}_Z(g \ s) \sqsubseteq_S h \ ps \end{array}$$

for all states s and property states ps. Thus the predicate says that if ps describes the sign of the variables of s then the sign of $g\ s$ will be described by $h\ ps$.

Exercise 2.17 Prove that for all arithmetic expressions $a \in$ **Aexp** we have

$$\mathcal{A}[\![a]\!] \ \mathsf{safe}_A \ \mathcal{SA}[\![a]\!]. \qquad \square$$

To express the safety of the analysis of boolean expressions we shall introduce a relation

$$g \ \mathsf{safe}_B \ h$$

for expressing when the analysis h: **PState** $\rightarrow$ **TT** is *safe with respect to* the semantics g: **State** $\rightarrow$ **T**. It is defined by

$$\begin{array}{l} \mathsf{abs}(s) \sqsubseteq_{PS} \ ps \\ \Downarrow \\ \mathsf{abs}_T(g \ s) \sqsubseteq_T h \ ps \end{array}$$

for all states s and property states ps. We have

Exercise 2.18 Prove that for all arithmetic expressions $b \in$ **Bexp** we have

$$\mathcal{B}[\![b]\!] \ \mathsf{safe}_B \ \mathcal{SB}[\![b]\!] \qquad \square$$

Statements

The safety of the analysis of statements will express that if the signs of the initial state is described by some property state then the signs of the final state (if ever reached) will be described by the property state obtained by applying the analysis to the initial property state. We shall formalize this by defining a relation

$$g \ \mathsf{safe}_S \ h$$

between a function g: **State** $\hookrightarrow$ **State**, perhaps of the form $\mathcal{S}_{ds}[\![S]\!]$ for some $S \in$ **Stm**, and another function h: **PState** $\rightarrow$ **PState**, perhaps of the form $\mathcal{SS}[\![S]\!]$ for some $S \in$ **Stm**. The formal definition amounts to

$$\begin{array}{l} \mathsf{abs}(s) \sqsubseteq_{PS} \ ps \text{ and } g \ s \neq \underline{\text{undef}} \\ \Downarrow \\ \mathsf{abs}(g \ s) \sqsubseteq_{PS} \ h \ ps \end{array}$$

for all states $s \in$ **State** and all property states $ps \in$ **PState**.

We may then formulate the desired relationship between the semantics and the analysis as follows:

Theorem 2.19 For all statements S of **While**: $\mathcal{S}_{ds}[\![S]\!] \ \mathsf{safe}_S \ \mathcal{SS}[\![S]\!]$.

Before conducting the proof we need to establish some properties of the auxiliary operations composition and conditional.

Lemma 2.20 Let g_1, g_2: **State** $\hookrightarrow$ **State** and h_1, h_2: **PState** $\rightarrow$ **PState**. Then

$$g_1 \ \mathsf{safe}_S \ h_1 \text{ and } g_2 \ \mathsf{safe}_S \ h_2 \text{ imply } g_2 \circ g_1 \ \mathsf{safe}_S \ h_2 \circ h_1$$

Proof Let s and ps be such that

$$\mathsf{abs}(s) \sqsubseteq_{PS} ps \text{ and } (g_2 \circ g_1)s \neq \underline{\text{undef}}$$

Then $g_1 \ s \neq \underline{\text{undef}}$ must be the case and from the assumption $g_1 \ \mathsf{safe}_S \ h_1$ we then get

$$\mathsf{abs}(g_1 \ s) \sqsubseteq_{PS} \ h_1 \ ps$$

Since $g_2 \ (g_1 \ s) \neq \underline{\text{undef}}$ we use the assumption $g_2 \ \mathsf{safe}_S \ h_2$ to get

$$\mathsf{abs}(g_2 \ (g_1 \ s)) \sqsubseteq_{PS} \ h_2(h_1 \ ps)$$

and we have completed the proof. $\square$

Lemma 2.21 Assume that g_1, g_2: **State** $\hookrightarrow$ **State**, and g: **State** $\rightarrow$ **T** and that h_1, h_2: **PState** $\rightarrow$ **PState** and f: **PState** $\rightarrow$ **TT**. Then

$$g \ \mathsf{safe}_B\ f,\ g_1\ \mathsf{safe}_S\ h_1 \text{ and } g_2\ \mathsf{safe}_S\ h_2 \text{ imply}$$

$$\mathsf{cond}(g,\ g_1,\ g_2)\ \mathsf{safe}_S\ \mathsf{cond}_S(f,\ h_1,\ h_2)$$

Proof Let s and ps be such that

$$\mathsf{abs}(s) \sqsubseteq_{PS}\ ps \text{ and } \mathsf{cond}(g,\ g_1,\ g_2)\ s \neq \underline{\text{undef}}$$

We shall now proceed by a case analysis on $g\ s$. First assume that $g\ s =$ **tt**. It must be the case that $g_1\ s \neq \underline{\text{undef}}$ so from $g_1\ \mathsf{safe}_S\ h_1$ we get

$$\mathsf{abs}(g_1\ s) \sqsubseteq_{PS}\ h_1\ ps$$

From $g\ \mathsf{safe}_B\ f$ we get that

$$\mathsf{abs}_T(g\ s) \sqsubseteq_T\ f\ ps$$

and thereby $\text{TT} \sqsubseteq_T f\ ps$. Since $h_1\ ps\ \sqsubseteq_{PS}\ h_1\ ps \sqcup_{PS} h_2\ ps$ we get the required result both when $f\ ps = \text{TT}$ and when $f\ ps = \text{ANY}$. The case where $g\ s =$ **ff** is similar. $\square$

We now have the apparatus needed to show the safety of $\mathcal{SS}$:

Proof (of Theorem 2.19) We shall show that $\mathcal{S}_{ds}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S]\!]$ and we proceed by structural induction on the statement S.

The case $x := a$: Let s and ps be such that

$$\mathsf{abs}(s) \sqsubseteq_{PS}\ ps \text{ and } \mathcal{S}_{ds}[\![x := a]\!]\ s \neq \underline{\text{undef}}$$

We have to show that

$$\mathsf{abs}(\mathcal{S}_{ds}[\![x := a]\!]\ s) \sqsubseteq_{PS} \mathcal{SS}[\![x := a]\!]ps$$

that is

$$\mathsf{abs}_Z((\mathcal{S}_{ds}[\![x := a]\!]\ s)\ y) \sqsubseteq_S (\mathcal{SS}[\![x := a]\!]ps)\ y$$

for all $y \in$ **Var**. If $y \neq x$ then it is immediate from the assumption $\mathsf{abs}(s)$ $\sqsubseteq_{PS}\ ps$. If $y = x$ then we use that Exercise 2.17 gives

$$\mathsf{abs}(\mathcal{A}[\![a]\!]\ s) \sqsubseteq_S \mathcal{SA}[\![a]\!]\ ps$$

Hence we have the required relationship.

The case `skip`: Straightforward.

The case S_1;S_2: The induction hypothesis applied to S_1 and S_2 gives

$$\mathcal{S}_{ds}[\![S_1]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S_1]\!] \text{ and } \mathcal{S}_{ds}[\![S_2]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S_2]\!]$$

The desired result

$$\mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S_2]\!] \circ \mathcal{SS}[\![S_1]\!]$$

follows directly from Lemma 2.20.

The case `if` b `then` S_1 `else` S_2: From Exercise 2.18 we have

$$\mathcal{B}[\![b]\!]\ \mathsf{safe}_B\ \mathcal{SB}[\![b]\!]$$

and the induction hypothesis applied to S_1 and S_2 gives

$$\mathcal{S}_{ds}[\![S_1]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S_1]\!] \text{ and } \mathcal{S}_{ds}[\![S_2]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S_2]\!]$$

The desired result

$$\mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!])\ \mathsf{safe}_S\ \mathsf{cond}_S(\mathcal{SB}[\![b]\!], \mathcal{SS}[\![S_1]\!], \mathcal{SS}[\![S_2]\!])$$

then follows directly from Lemma 2.21.

The case `while` b `do` S: We must prove that

$$\mathsf{FIX}(G)\ \mathsf{safe}_S\ \mathsf{FIX}(H)$$

where

$$G\ g = \mathsf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], \mathsf{id})$$
$$H\ h = \mathsf{cond}_S(\mathcal{SB}[\![b]\!], h \circ \mathcal{SS}[\![S]\!], \mathsf{id})$$

To do this we recall from Chapter 4 of [NN] the definition of the least fixed points:

$$\mathsf{FIX}\ G = \bigsqcup \{G^n\, g_0 \mid n \ \geq\ 0\} \text{ where } g_0\ s = \underline{\text{undef}} \text{ for all } s$$
$$\mathsf{FIX}\ H = \bigsqcup \{H^n\, h_0 \mid n \ \geq\ 0\} \text{ where } h_0\ ps = \text{INIT} \text{ for all } ps$$

The proof proceeds in two stages. We begin by proving that

$$G^n\ g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H \text{ for all } n \qquad (*)$$

and then

$$\mathsf{FIX}\ G\ \mathsf{safe}_S\ \mathsf{FIX}\ H \qquad (**)$$

We prove (*) by induction on n. For the base case we observe that

$$g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

holds trivially since $g_0\ s = \underline{\text{undef}}$ for all states s. For the induction step we assume that

$$G^n\ g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

and we shall prove the result for $n+1$. We have

$$\mathcal{B}[\![b]\!]\ \mathsf{safe}_B\ \mathcal{SB}[\![b]\!]$$

from Exercise 2.18,

$$\mathcal{S}_{ds}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S]\!]$$

from the induction hypothesis applied to the body of the `while`-loop, and it is clear that

$$\mathsf{id}\ \mathsf{safe}_S\ \mathsf{id}$$

We then obtain

$$\begin{aligned}&\mathsf{cond}(\mathcal{B}[\![b]\!], (G^n\ g_0) \circ \mathcal{S}_{ds}[\![S]\!], \mathsf{id})\ \mathsf{safe}_S\\ &\qquad \mathsf{cond}_S(\mathcal{SB}[\![b]\!], (\mathsf{FIX}\ H) \circ \mathcal{SS}[\![S]\!], \mathsf{id})\end{aligned}$$

from Lemmas 2.20 and 2.21 and this is indeed the desired result since the right-hand side amounts to $H\ (\mathsf{FIX}\ H)$ which equals $\mathsf{FIX}\ H$.

Finally we must show (**). This amounts to showing

$$(\textstyle\bigsqcup Y)\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

where $Y = \{\ G^n g_0 \mid n\ \geq\ 0\ \}$. So assume that

$$\mathsf{abs}(s) \sqsubseteq_{PS} \; ps \text{ and } (\bigsqcup Y)\; s \neq \underline{\text{undef}}$$

By [NN, Lemma 4.25] we have

$$\mathsf{graph}(\bigsqcup Y) = \bigcup \{\; \mathsf{graph}\; g \mid g \in Y \;\}$$

and $(\bigsqcup Y)\; s \neq \underline{\text{undef}}$ therefore shows the existence of an element g in Y such that $g\; s \neq \underline{\text{undef}}$ and $(\bigsqcup Y)\; s = g\; s$. Since g safe_S $\mathsf{FIX}\; H$ holds for all $g \in Y$ by (*) we get that

$$\mathsf{abs}(g\; s) \sqsubseteq_{PS} (\mathsf{FIX}\; H)\; ps$$

and therefore $\mathsf{abs}((\bigsqcup Y) s) \sqsubseteq_{PS} (\mathsf{FIX}\; H)\; ps$ as required. □

Exercise 2.22 Extend the proof of Theorem 2.19 to incorporate the safety of the analysis developed for `repeat` S `until` b in Exercise 2.15. □

Exercise 2.23 Prove that the constant propagation analysis specified in Exercise 2.8 is safe. That is show that

$$\mathcal{A}[\![a]\!] \; \mathsf{safe}_A \; \mathcal{CA}[\![a]\!]$$
$$\mathcal{B}[\![b]\!] \; \mathsf{safe}_B \; \mathcal{CB}[\![b]\!]$$
$$\mathcal{S}_{ds}[\![a]\!] \; \mathsf{safe}_S \; \mathcal{CS}[\![S]\!]$$

for appropriately defined predicates safe_A, safe_B and safe_S. □

2.4 Application of the Analysis

The detection of signs analysis can be used to predict the values of tests in conditionals and loops and thereby they may be used to facilitate certain program transformations. Some program transformations have a *global* nature as for example

Replace	`if` b `then` S_1 `else` S_2
by	S_1
when	$\mathcal{SB}[\![b]\!]\; \text{TOP} = \text{TT}$

where we have written TOP for the property state that maps all variables to ANY. The condition $\mathcal{SB}[\![b]\!]$ TOP $=$ TT will only be satisfied when $\mathcal{B}[\![b]\!]\ s =$ **tt** for all states s so the transformation can be used rather seldom. Other transformations take the *context* into account and we may use the property states to describe the contexts. So we may extend the above format to:

In context	ps
replace	`if` b `then` S_1 `else` S_2
by	S_1
when	$\mathcal{SB}[\![b]\!]ps = \text{TT}$

We shall formalise these transformations as a transition system. The transitions have the form

$$ps \vdash S \leadsto S'$$

meaning that in the context described by ps the statement S can be replaced by S'. So the above transformation rule can be formulated as

$$ps \vdash \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \leadsto S_1, \text{ if } \mathcal{SB}[\![b]\!]ps = \text{TT}$$

where the side condition expresses when the rule is applicable. The dual transformation rule is

$$ps \vdash \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \leadsto S_2, \text{ if } \mathcal{SB}[\![b]\!]ps = \text{FF}$$

We might also want to transform inside composite statements like $S_1; S_2$. This is expressed by the rule

$$\frac{ps \vdash S_1 \leadsto S_1', \qquad (\mathcal{SS}[\![S_1]\!]\ ps) \vdash S_2 \leadsto S_2'}{ps \vdash S_1; S_2 \leadsto S_1'; S_2'}$$

Combined with the trivial transformation rule

$$ps \vdash S \leadsto S$$

this opens up for the possibility of only transforming parts of the statement.

In general a transformation $ps \vdash S \leadsto S'$ is (weakly) *valid* if

$$\begin{array}{l} \mathsf{abs}(s) \sqsubseteq_{PS} ps \text{ and } \mathcal{S}_{ds}[\![S]\!]s \neq \underline{\text{undef}} \\ \Downarrow \\ \mathcal{S}_{ds}[\![S]\!]s = \mathcal{S}_{ds}[\![S']\!]s \end{array}$$

for all states s. This is a *weak* notion of validity because a transformation as

$$\textsc{top} \vdash \texttt{while true do skip} \rightsquigarrow \texttt{skip}$$

is valid even though it allows us to replace a looping statement with one that always terminates.

Lemma 2.24 The transformation rule

$$ps \vdash \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \rightsquigarrow S_1$$

is valid provided that $\mathcal{SB}[\![b]\!]ps = \textsc{tt}$.

Proof Assume that

$$\mathsf{abs}(s) \sqsubseteq_{PS} ps \text{ and } \mathcal{S}_{ds}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\ s \neq \underline{\text{undef}}.$$

From $\mathcal{SB}[\![b]\!]ps = \textsc{tt}$, $\mathsf{abs}(s) \sqsubseteq_{PS} ps$, and Exercise 2.18 we get that

$$\mathsf{abs}_T(\mathcal{B}[\![b]\!]s) \sqsubseteq_T \textsc{tt}$$

and thereby $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ since $\mathcal{B}[\![b]\!]$ is a total function. From the definition of $\mathcal{S}_{ds}$ in Chapter 4 of [NN] we get

$$\mathcal{S}_{ds}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\ s = \mathcal{S}_{ds}[\![S_1]\!]\ s$$

and this is the required result. □

Exercise 2.25 Prove that the transformation rule for $S_1; S_2$ is valid. □

Exercise 2.26 Prove that the transformation rule

$$\frac{ps \vdash S_1 \rightsquigarrow S_1', \qquad (\mathcal{SS}[\![S_1']\!]\ ps) \vdash S_2 \rightsquigarrow S_2'}{ps \vdash S_1; S_2 \rightsquigarrow S_1'; S_2'}$$

is valid. Note that it differs from the rule given earlier in the second premise where the transformed statement is analysed rather than the original. □

Exercise 2.27 Suggest a transformation rule for assignment and prove that it is valid. □

Exercise 2.28 Suggest a transformation rule that allows transformations inside the branches of a conditional and prove that it is valid. □

Exercise 2.29 ** Try to develop a transformation rule that allows transformations inside the body of a `while`-loop. □

Chapter 3

Implementation of Analyses

In this chapter we consider the problem of computing fixed points in program analysis. The whole purpose of program analysis is to get information about programs without actually running them and it is important that the analyses always terminate. In general, the analysis of a recursive (or iterative) program will itself be recursively defined and it is therefore important to "solve" this recursion such that termination is ensured.

In general the setting is as follows: To each program the analysis associates an element h of a complete lattice $(A \to B, \sqsubseteq)$ of abstract values. In the case of an iterative construct as the `while`-loop the value h is determined as the least fixed point, $\mathsf{FIX}\ H$, of a continuous functional $H : (A \to B) \to (A \to B)$. Formally, the fixed point of H is given by

$$\mathsf{FIX}\ H = \bigsqcup\{H^i\bot \mid i \ \geq \ 0\}$$

where $\bot$ is the least element of $A \to B$ and $\bigsqcup$ is the least upper bound operation on $A \to B$. It is well-known that the iterands $\{H^i\bot \mid i \ \geq \ 0\}$ is a chain in $A \to B$ and in Exercise 2.7 we have shown that

$$\text{if } H^k\bot = H^{k+1}\bot \text{ for some } k \ \geq \ 0 \text{ then } \mathsf{FIX}\ H = H^k\bot$$

So the obvious algorithm for computing $\mathsf{FIX}\ H$ will be to determine the iterands $H^0\bot,\ H^1\bot, \cdots$ one after the other while testing for stabilization, i.e. equality with the predecessor. When $A \to B$ is finite this procedure is guaranteed to terminate. The cost of this algorithm depends on

- the number k of iterations needed before stabilization,
- the cost of comparing two iterands, and

- the cost of computing a new iterand.

We shall now study how to minimize the cost of the above algorithm. Most of our efforts are spent on minimizing the number k.

We shall assume that A and B are finite complete lattices and we shall consider three versions of the framework:

- the *general framework* where functions of $A \to B$ only are required to be total; this is written $A \to_t B$.
- the *monotone framework* where functions of $A \to B$ must be monotone; this is written $A \to_m B$.
- the *completely additive framework* where functions of $A \to B$ must be strict and additive; this is written $A \to_{sa} B$.

We give precise bounds on the number k of iterations needed to compute the fixed point of an arbitrary continuous functional H.

For the detection of signs analysis and many other program analyses A and B will both be **PState**. Since a given program always mentions a finite number of variables we can assume that **Var** is finite so **PState** = **Var** $\to$ **P** is a function space with a finite domain. In the more general development we shall therefore pay special attention to the case where $A = B = S \to L$ for some non-empty set S with p elements and some finite complete lattice $(L, \sqsubseteq)$. We shall show that the number k of iterations needed to compute the fixed point is at most

- *exponential* in p for the general and the monotone frameworks, and
- *quadratic* in p for the completely additive framework.

The above results hold for arbitrary continuous functionals H. A special case is where H is in iterative form:

> H is in *iterative form* if it is of the form $H\ h = f \sqcup (h \circ g)$

For strict and additive functions f and g we then show that k is at most

- *linear* in p, and furthermore
- the fixed point can be computed *pointwise*.

This result is of particular interest for the analysis of **While** programs since the functional obtained for the `while`-loop often can be written in this form.

3.1 The general and monotone frameworks

We shall first introduce some notation. Let $(D, \sqsubseteq)$ be a *finite complete lattice*, that is

- $\sqsubseteq$ is a partial order on D, and
- each subset Y of D has a least upper bound in D denoted $\bigsqcup Y$.

When $d \sqsubseteq d' \wedge d \neq d'$ we simply write $d \sqsubset d'$. Next we write

$$
\begin{array}{lcl}
\mathbf{C}\ D & : & \text{for the cardinality of } D \\
\mathbf{H}\ D & : & \text{for the maximal length of chains in } D
\end{array}
$$

where a chain $\{d_0, d_1, \cdots, d_k\}$ has length k if it contains $k+1$ distinct elements. As an example **Sign** has cardinality 8 and height 3 whereas **TT** has cardinality 4 and height 2. For a complete lattice of the form $S \to L$ we have

---

Fact 3.1 $\mathbf{C}(S \to L) = (\mathbf{C}\ L)^p$ and $\mathbf{H}(S \to L) = p \cdot (\mathbf{H}\ L)$ for $p \geq 1$ being the cardinality of S.

Thus for the detection of signs analysis of the factorial program we have $\mathbf{C}$ **PState** $= 64$ and $\mathbf{H}$ **PState** $= 6$ because the program contain only two variables.

In the general framework we have

Proposition 3.2 $\mathbf{H}(A \to_t B) \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$.

Proof Let $h_i : A \to_t B$ and assume that

$$h_0 \sqsubset h_1 \sqsubset \cdots \sqsubset h_k$$

From $h_i \sqsubset h_{i+1}$ we get that there exists $w \in A$ such that $h_i\ w \sqsubset h_{i+1}\ w$ because the ordering on $A \to_t B$ is defined componentwise. There are at most $(\mathbf{C}\ A)$ choices of w and each w can occur at most $(\mathbf{H}\ B)$ times. Thus

$$k \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$$

as was to be shown. □

Any monotone function is a total function so Proposition 3.2 yields:

Corollary 3.3 Corollary 3.3: $\mathbf{H}(A \to_m B) \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$.

We shall now apply Proposition 3.2 to the special chains obtained when computing fixed points:

Theorem 3.4 In the general framework any continuous functional

$$H : (A \to_t B) \to (A \to_t B)$$

satisfies $\mathsf{FIX}\ H = H^k \bot$ for

$$k = (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$$

This result carries over to the monotone framework as well. When $A = B = S \to L$ we have $k = (\mathbf{C}\ L)^p \cdot p \cdot (\mathbf{H}\ L)$ where p is the cardinality of S.

Proof Consider the chain

$$H^0 \bot \sqsubseteq H^1 \bot \sqsubseteq \cdots$$

Since $A \to_t B$ is a finite complete lattice it cannot be the case that all $H^i \bot$ are distinct. Let k' be the minimal index for which $H^{k'} \bot = H^{k'+1} \bot$. Then

$$H^0 \bot \sqsubset H^1 \bot \sqsubset \cdots \sqsubset H^{k'} \bot$$

Using Proposition 3.2 we then get that $k' \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$, i.e. $k' \leq k$. Since $H^{k'} \bot = \mathsf{FIX}\ H$ and $H^{k'} \bot \sqsubseteq H^k \bot \sqsubseteq \mathsf{FIX}\ H$ we get $\mathsf{FIX}\ H = H^k \bot$ as required. □

Note that by finiteness of A and B the continuity of H simply amounts to monotonicity of H.

Example 3.5 The detection of signs analysis of the factorial program gives rise to a continuous functional

$$H : (\mathbf{PState} \to_t \mathbf{PState}) \to (\mathbf{PState} \to_t \mathbf{PState})$$

Now **C PState** $= 64$ and **H PState** $= 6$ because the factorial program only contains two variables. So, according to the theorem, at most $64 \cdot 6 = 384$ iterations are needed to determine the fixed point. However, the simple calculation of Example 2.6 shows that the fixed point is obtained already after the *second* iteration! Thus our bound is very pessimistic. □

3.2 The completely additive framework

The reason why the upper bound determined in Theorem 3.4 is so imprecise is that we consider *all* functions in **PState** →**PState** and do not exploit any special properties of the functions $H^n\bot$, such as continuity. To obtain a better bound we shall exploit properties of the $\mathcal{SS}[\![S]\!]$ analysis functions.

We shall assume that the functions of interest are strict and additive; by *strictness* of a function h we mean that

$$h \perp = \perp$$

and by *additivity* that

$$h(d_1 \sqcup d_2) = (h\ d_1) \sqcup (h\ d_2)$$

Since the complete lattices considered are all finite it follows that a strict and additive function h is also *completely additive*, that is

$$h(\bigsqcup Y) = \bigsqcup\{\ h\ d \mid d\ \in\ Y\}$$

for all subsets Y.

Exercise 3.2.1 Consider the detection of signs analysis and the claims: (1) each $\mathcal{SA}[\![a]\!]$ is strict; (2) each $\mathcal{SB}[\![b]\!]$ is strict; (3) each $\mathcal{SS}[\![S]\!]$ is strict; (4) each $\mathcal{SA}[\![a]\!]$ is additive; (5) each $\mathcal{SB}[\![b]\!]$ is additive; (6) each $\mathcal{SS}[\![S]\!]$ is additive. Determine the truth or falsity of each of these claims.

An element d of a complete lattice $(D, \sqsubseteq)$ is called *join-irreducible* if for all $d_1, d_2\ \in\ L$:

$$d = d_1 \sqcup d_2 \text{ implies } d = d_1 \text{ or } d = d_2$$

As an example **Sign** has four join-irreducible elements: NONE, NEG, ZERO and POS. The element NON-POS is not join-irreducible since it is the least upper bound of ZERO and NEG but is is equal to none of them.

From the definition it follows that the least element $\perp$ of D is always join-irreducible but we shall be more interested in the non-trivial join-irreducible elements, i.e. those that are not $\perp$. To this end we shall write

RJC L : for the number of non-bottom join-irreducible elements of L.

We thus have **RJC Sign** $= 3$ and **RJC TT** $= 2$.

Fact 3.6 $\mathbf{RJC}(S \to L) = p \cdot (\mathbf{RJC}\ L)$ where p is the cardinality of S.

Proof The join-irreducible elements of $S \to L$ are those functions h that map all but one element of S to $\perp$ and one element to a join-irreducible element of L. $\square$

Lemma 3.7 If $(L, \sqsubseteq)$ is a finite complete lattice we have

$$w = \bigsqcup\{\, x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \perp\}$$

for all $w \in L$.

Proof Assume by way of contradiction that the claim of the lemma is false. Let $W \subseteq L$ be the set of $w \in L$ for which the condition fails. Since W is finite and non-empty it has a minimal element w. From $w \in W$ it follows that w is not join-irreducible. Hence there exist w_1 and w_2 such that

$$w = w_1 \sqcup w_2,\ w \neq w_1,\ w \neq w_2$$

It follows that $w_1 \sqsubset w$, $w_2 \sqsubset w$ and by choice of w that $w_1 \notin W$ and $w_2 \notin W$. We may then calculate

$$\begin{array}{rcl}
w & = & w_1 \sqcup w_2 \\
 & = & \bigsqcup\{\, x \mid x \sqsubseteq w_1,\ x \text{ is join-irreducible and } x \neq \perp\} \\
 & & \sqcup \bigsqcup\{\, x \mid x \sqsubseteq w_2,\ x \text{ is join-irreducible and } x \neq \perp\} \\
 & = & \bigsqcup\{\, x \mid (x \sqsubseteq w_1 \text{ or } x \sqsubseteq w_2),\ x \text{ is join-irreducible and } x \neq \perp\} \\
 & \sqsubseteq & \bigsqcup\{\, x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \perp\} \\
 & \sqsubseteq & w
\end{array}$$

This shows that

$$w = \bigsqcup\{\ x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x\ \neq\ \bot\}$$

and contradicts $w \in W$. Hence $W = \emptyset$ and the claim of the lemma holds. □

In the completely additive framework we have

Proposition 3.8 $\mathbf{H}(A \rightarrow_{sa} B) \leq (\mathbf{RJC}\ A) \cdot (\mathbf{H}\ B)$.

Proof The proof is a refinement of that of Proposition 3.2. So we begin by assuming that $h_i\ \in\ A \rightarrow_{sa} B$ and that

$$h_0 \sqsubset h_1 \sqsubset \cdots \sqsubset h_k$$

As in the proof of Proposition 3.2 we get an element w such that $h_i\ w \sqsubset h_{i+1}\ w$ for each $h_i \sqsubset h_{i+1}$. The element w is an arbitrary element of A so in the proof of Proposition 3.2 there was $(\mathbf{C}\ A)$ choices for w. We shall now show that w can be chosen as a non-trivial join-irreducible element of A thereby reducing the number of choices to $(\mathbf{RJC}\ A)$. Calculations similar to those in the proof of Proposition 3.2 will then give the required upper bound on k, i.e. $k \leq (\mathbf{RJC}\ A) \cdot (\mathbf{H}\ B)$.

The element w satisfies $h_i\ w \sqsubset h_{i+1}\ w$. By Lemma 3.7 we have

$$w = \bigsqcup\{\ x \mid x \sqsubseteq w,\ x \text{ is a join-irreducible and } x\ \neq\ \bot\}$$

From the strictness and additivity of h_i and h_{i+1} we get

$$\begin{array}{lcl} h_i\ w & = & \bigsqcup\{\ h_i\ x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x\ \neq\ \bot\} \\ h_{i+1}\ w & = & \bigsqcup\{\ h_{i+1}\ x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x\ \neq\ \bot\} \end{array}$$

It cannot be the case that $h_i\ x = h_{i+1}\ x$ for all non-bottom join-irreducible elements x of A since then $h_i\ w = h_{i+1}\ w$. So let x be a non-bottom join-irreducible element where $h_i\ x \sqsubset h_{i+1}\ x$. Then there will only be $(\mathbf{RJC}\ A)$ choices for x and this completes the proof. □

We can now apply Proposition 3.8 to the special chains obtained when computing fixed points:

Theorem 3.9 In the completely additive framework any continuous functional

$$H : (A \to_{sa} B) \to (A \to_{sa} B)$$

satisfies $\mathsf{FIX}\ H = H^k\bot$ for

$$k = (\mathbf{RJC}\ A) \cdot (\mathbf{H}\ B)$$

When $A = B = S \to L$ we have $k = p \cdot (\mathbf{RJC}\ L) \cdot p \cdot (\mathbf{H}\ L)$ where p is the cardinality of S.

Proof Analoguous to the proof of Theorem 3.4. □

The equality test between the iterands $H^0\bot, H^1\bot, \cdots$ can be simplified in this framework. To see this consider two functions $h_1, h_2 \in A \to_{sa} B$. Then

$$h_1 = h_2$$

if and only if

$$h_1\ x = h_2\ x \text{ for all non-trivial join-irreducible elements } x \text{ of } A.$$

3.3 Iterative program schemes

The upper bounds expressed by Theorems 3.4 and 3.9 are obtained without any assumptions about the functional H except that it is a continuous function over the relevant lattices. In this section we shall restrict the form of H.

For iterative programs as e.g. a `while`-loop the functional H will typically have the form

$$H\ h = f \sqcup (h \circ g)$$

Since our aim is to further improve the bound of the previous section we shall assume that f and g are strict and additive functions. Then also the iterands $H^i\bot$ will be strict and additive.

Exercise 3.3.1 The functionals obtained by analysing the `while`-loop in the detection of signs analysis can be written in this form. To see this define the auxiliary functions $\mathsf{f\text{-}func}$ and $\mathsf{g\text{-}func}$ by

$$\mathsf{f\text{-}func}(f)ps \;=\; \begin{cases} \text{INIT} & \text{if } f\ ps = \text{NONE or } f\ ps = \text{TT} \\ ps & \text{otherwise} \end{cases}$$

$$\mathsf{g\text{-}func}(f,g)ps \;=\; \begin{cases} \text{INIT} & \text{if } f\ ps = \text{NONE or } f\ ps = \text{FF} \\ g\ ps & \text{otherwise} \end{cases}$$

Show that the functional H obtained from `while` b `do` S can be written as $H\ h = \mathsf{f\text{-}func}(\mathcal{SB}[\![b]\!]) \sqcup h\circ\mathsf{g\text{-}func}(\mathcal{SB}[\![b]\!], \mathcal{SS}[\![S]\!])$.

The first result is a refinement of Theorem 3.9:

Theorem 3.10 Let $f \in A \to_{sa} B$ and $g \in A \to_{sa} A$ be given and define

$$H\ h = f \sqcup (h \circ g)$$

Then $H : (A \to_{sa} B) \to (A \to_{sa} B)$ is a continuous functional and taking

$$k = (\mathbf{H}\ A)$$

will ensure

$$\mathsf{FIX}\ H = H^k \bot$$

When $A = B = S \to L$ we have $k = p \cdot (\mathbf{H}\ L)$ where p is the cardinality of S.

Writing $H_0\ h = \mathtt{id} \sqcup (h \circ g)$ and taking k_0 to satisfy

$$H_0^{k_0} \bot\ w = H_0^{k_0+1} \bot\ w$$

we have

$$\mathsf{FIX}\ H\ w = H^{k_0} \bot\ w = f(H_0^{k_0} \bot\ w)$$

In other words, fixed points of H_0 may be computed pointwise.

Basically this result says that in order to compute $\mathsf{FIX}\ H$ on a particular value w it is sufficient to determine the values of the iterands $H_0^i\bot$ at w and then compare these values. So rather than having to test the extensional equality of two functions on a set of arguments we only need to test the equality of two function values. Furthermore, the theorem states that this test has to be performed at most a linear number of times.

To prove the theorem we need a couple of lemmas:

Lemma 3.11 Let $H\ h = f \sqcup (h \circ g)$ for $f \in A \rightarrow_{sa} B$ and $g \in A \rightarrow_{sa} A$. Then for $i \geq 0$ we have

$$H^{i+1}\bot = \bigsqcup\{\ f \circ g^j \mid 0 \leq j \leq i\}.$$

Proof We proceed by numerical induction on i. If $i = 0$ then the result is immediate as $H^1\bot = f \sqcup (\bot \circ g) = f = \bigsqcup\{\ f \circ g^j \mid 0 \leq j \leq 0\}$. For the induction step we calculate:

$$\begin{array}{rcl} H^{i+2}\bot & = & f \sqcup (H^{i+1}\bot) \circ g \\ & = & f \sqcup (\bigsqcup\{\ f \circ g^j \mid 0 \leq j \leq i\}) \circ g \\ & = & \bigsqcup\{\ f \circ g^j \mid 0 \leq j \leq i+1\} \end{array}$$

where the last equality follows from the pointwise definition of $\bigsqcup$ on $A \rightarrow_{sa} B$. $\square$

Lemma 3.12 Let $H\ h = f \sqcup (h \circ g)$ for $f \in A \rightarrow_{sa} B$ and $g \in A \rightarrow_{sa} A$. Then

$$\mathsf{FIX}\ H = f \circ (\mathsf{FIX}\ H_0) \text{ and } H^k\bot = f \circ H_0^k\bot$$

where $H_0\ h = \mathtt{id} \sqcup (h \circ g)$.

Proof We shall first prove that

$$H^i\bot = f \circ H_0^i\bot \text{ for } i \geq 0.$$

The case $i = 0$ is immediate because f is strict. So assume that $i > 0$. We shall then apply Lemma 3.11 to H and H_0 and get

$$\begin{array}{lcl} H^i\bot & = & \bigsqcup\{\ f\circ g^j \mid 0\leq j\leq i-1\} \\ H_0^i\bot & = & \bigsqcup\{\ g^j \mid 0\leq j\leq i-1\} \end{array}$$

Since f is additive we get

$$H^i\bot = f\circ\bigsqcup\{\ g^j \mid 0\leq j\leq i-1\} = f\circ H_0^i\bot$$

as required.

We now have

$$\begin{array}{lcl} \mathsf{FIX}\ H & = & \bigsqcup\{\ H^i\bot \mid i\ \geq\ 0\} \\ & = & \bigsqcup\{\ f\circ H_0^i\bot \mid i\ \geq\ 0\} \\ & = & f\circ\bigsqcup\{\ H_0^i\bot \mid i\ \geq\ 0\} \\ & = & f\circ(\mathsf{FIX}\ H_0) \end{array}$$

where the third equality uses the complete additivity of f. □

Proof of Theorem 3.10: We shall first prove that if $H_0^{k_0}\bot\ w = H_0^{k_0+1}\bot\ w$ then $\mathsf{FIX}\ H\ w = f(H_0^{k_0}\bot\ w)$. This is done in two stages.

First assume that $k_0 = 0$. Then $H_0^0\bot\ w = H_0^1\bot\ w$ amounts to $\bot = w$. Using Lemma 3.11 and the strictness of g we get

$$H_0^{i+1}\bot\ \bot = \bigsqcup\{\ g^j\bot \mid 0\leq j\leq i\} = \bot$$

for $i\ \geq\ 0$ and thereby $\mathsf{FIX}\ H_0\ \bot = \bot$. But then Lemma 3.12 gives

$$\mathsf{FIX}\ H\ \bot = f(\mathsf{FIX}\ H_0\ \bot) = f\bot = f(H_0^0\bot\ \bot)$$

as required.

Secondly assume that $k_0 > 0$. From $H_0^{k_0}\bot\ w = H_0^{k_0+1}\bot\ w$ we get, using Lemma 3.11, that

$$\bigsqcup\{\ g^j\ w \mid 0\leq j< k_0\} = \bigsqcup\{\ g^j\ w \mid 0\leq j\leq k_0\}$$

This means that

$$g^{k_0}\ w \sqsubseteq \bigsqcup\{\ g^j\ w \mid 0\leq j< k_0\}$$

We shall now prove that for all $l\ \geq\ 0$

$$g^{k_0+l}\ w \sqsubseteq \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\} \tag{$*$}$$

We have already established the basis $l = 0$. For the induction step we get

$$\begin{array}{rcl} g^{k_0+l+1}\ w &=& g(g^{k_0+l}\ w) \\ &\sqsubseteq& g(\bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\}) \\ &=& \bigsqcup\{\ g^j\ w \mid 1 \leq j \leq k_0\} \\ &\sqsubseteq& \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\} \end{array}$$

where we have used the additivity of g. This proves $(*)$. Using Lemma 3.11 and $(*)$ we get

$$\begin{array}{rcl} H_0^{k_0+l}\bot\ w &=& \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0 + l\} \\ &=& \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\} \\ &=& H_0^{k_0}\bot\ w \end{array}$$

for all $l \geq 0$. This means that $\mathsf{FIX}\ H_0\ w = H_0^{k_0}\bot\ w$ and using Lemma 3.12 we get

$$\mathsf{FIX}\ H\ w = f(H_0^{k_0}\bot\ w)$$

as required.

To complete the proof of the theorem we have to show that one may take $k = (\mathbf{H}\ A)$. For this it suffices to show that one cannot have a chain

$$H_0^0\bot\ w \sqsubset H_0^1\bot\ w \sqsubset \cdots \sqsubset H_0^k\bot\ w \sqsubset H_0^{k+1}\bot\ w$$

in A. But this is immediate since $k + 1 > \mathbf{H}(A)$. $\square$

Chapter 4

More Program Analyses

The detection of signs analysis is an example of a *forward analysis*: the analysis of the program proceeds as the computation does, the only difference being that we compute with abstract properties rather than concrete values. Actually, many forward program analyses can be seen as a slight variation of the detection of signs analysis: the properties and the way we compute with them may be different but the overall approach will be the same. In this chapter we shall extract the overall approach and present it as a *general framework* where only a few parameters need to be specified in order to obtain the desired analysis.

4.1 The Framework

In general the *specification* of a program analysis falls into two parts. First we introduce the properties that the analysis operates on and next we specify the actual analysis for the three syntactic categories of **While**.

Properties and Property States

The *basic properties* are those of the numbers and the truth values. So the first step in specifying an analysis will be to define

P1: a complete lattice of properties of $\mathbf{Z}$: $(\mathbf{P}_Z, \sqsubseteq_Z)$

P2: a complete lattice of properties of $\mathbf{T}$: $(\mathbf{P}_T, \sqsubseteq_T)$

Example 4.1.1 In the case of the detection of signs analysis we have $\mathbf{P}_Z = \mathbf{Sign}$ and $\mathbf{P}_T = \mathbf{TT}$.

$$
\begin{array}{lcl}
\mathcal{FA}[\![n]\!]ps & = & \mathcal{FZ}[\![n]\!]\ ps \\
\mathcal{FA}[\![x]\!]ps & = & ps\ x \\
\mathcal{FA}[\![a_1 + a_2]\!]ps & = & \mathsf{add}_F\ (\mathcal{FA}[\![a_1]\!]ps,\ \mathcal{FA}[\![a_2]\!]ps) \\
\mathcal{FA}[\![a_1 \star a_2]\!]ps & = & \mathsf{mult}_F\ (\mathcal{FA}[\![a_1]\!]ps,\ \mathcal{FA}[\![a_2]\!]ps) \\
\mathcal{FA}[\![a_1 - a_2]\!]ps & = & \mathsf{sub}_F\ (\mathcal{FA}[\![a_1]\!]ps,\ \mathcal{FA}[\![a_2]\!]ps) \\
\\
\mathcal{FB}[\![\mathtt{true}]\!]ps & = & \mathcal{FT}[\![\mathtt{true}]\!]\ ps \\
\mathcal{FB}[\![\mathtt{false}]\!]ps & = & \mathcal{FT}[\![\mathtt{false}]\!]\ ps \\
\mathcal{FB}[\![a_1 = a_2]\!]ps & = & \mathsf{eq}_F\ (\mathcal{FA}[\![a_1]\!]ps,\ \mathcal{FA}[\![a_2]\!]ps) \\
\mathcal{FB}[\![a_1 \leq a_2]\!]ps & = & \mathsf{leq}_F\ (\mathcal{FA}[\![a_1]\!]ps,\ \mathcal{FA}[\![a_2]\!]ps) \\
\mathcal{FB}[\![\neg\ b]\!]ps & = & \mathsf{neg}_F\ (\mathcal{FB}[\![b]\!]ps) \\
\mathcal{FB}[\![b_1\ \wedge\ b_2]\!]ps & = & \mathsf{and}_F\ (\mathcal{FB}[\![b_1]\!]ps,\ \mathcal{FB}[\![b_2]\!]ps)
\end{array}
$$

Table 4.1: Forward analysis of expressions

The *property states* will then be defined as

$$\mathbf{PState} = \mathbf{Var} \to \mathbf{P}_Z$$

independently of the choice of $\mathbf{P}_Z$. The property states inherit the ordering of $\mathbf{P}_Z$ as indicated in Lemma 1.1 and will thus form a complete lattice. In particular, **PState** will have a least element which we call INIT.

Forward Analysis

In a forward analysis the computation proceeds much as in the direct style denotational semantics: given properties of the input the analysis will compute properties of the output. Thus the idea will be to replace the semantic functions

$$\mathcal{A}\colon \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Z}$$
$$\mathcal{B}\colon \mathbf{Bexp} \to \mathbf{State} \to \mathbf{T}$$
$$\mathcal{S}_{ds}\colon \mathbf{Stm} \to \mathbf{State} \hookrightarrow \mathbf{State}$$

with semantic functions that compute with properties rather than values:

$$\mathcal{FA}\colon \mathbf{Aexp} \to \mathbf{PState} \to \mathbf{P}_Z$$

$\mathcal{FB}$: **Bexp** $\rightarrow$ **PState** $\rightarrow$ $\mathbf{P}_T$

$\mathcal{FS}$: **Stm** $\rightarrow$ **PState** $\rightarrow$ **PState**

The semantic functions $\mathcal{FA}$ and $\mathcal{FB}$ are defined in Table 4.1. Whenever the direct style semantics performs computations involving numbers or truth values the analysis has to do something analogous depending on the actual choice of properties. We shall therefore assume that we have functions

F1: add_F: $\mathbf{P}_Z \times \mathbf{P}_Z \rightarrow \mathbf{P}_Z$

F2: mult_F: $\mathbf{P}_Z \times \mathbf{P}_Z \rightarrow \mathbf{P}_Z$

F3: sub_F: $\mathbf{P}_Z \times \mathbf{P}_Z \rightarrow \mathbf{P}_Z$

F4: eq_F: $\mathbf{P}_Z \times \mathbf{P}_Z \rightarrow \mathbf{P}_T$

F5: leq_F: $\mathbf{P}_Z \times \mathbf{P}_Z \rightarrow \mathbf{P}_T$

F6: neg_F: $\mathbf{P}_T \rightarrow \mathbf{P}_T$

F7: and_F: $\mathbf{P}_T \times \mathbf{P}_T \rightarrow \mathbf{P}_T$

describing how the analysis proceeds for the operators of arithmetic and boolean operators. Furthermore, we need a way of turning numbers and truth values into properties:

F8: $\mathcal{FZ}$: **Num** $\rightarrow$ **PState** $\rightarrow$ $\mathbf{P}_Z$

F9: $\mathcal{FT}$: $\{\mathtt{true}, \mathtt{false}\} \rightarrow$ **PState** $\rightarrow$ $\mathbf{P}_T$

Example 4.1.2 For the detection of signs analysis we have

$$\begin{array}{lcl}
\mathsf{add}_F\ (p_1, p_2) & = & p_1\ +_S\ p_2 \\
\mathsf{mult}_F\ (p_1, p_2) & = & p_1\ \star_S\ p_2 \\
\mathsf{sub}_F\ (p_1, p_2) & = & p_1\ -_S\ p_2 \\
\mathsf{eq}_F\ (p_1, p_2) & = & p_1\ =_S\ p_2 \\
\mathsf{leq}_F\ (p_1, p_2) & = & p_1\ \leq_S\ p_2 \\
\mathsf{neg}_F\ p & = & \neg_T\ p \\
\mathsf{and}_F\ (p_1, p_2) & = & p_1\ \wedge_T\ p_2
\end{array}$$

where $+_S$, $\star_S$, $-_S$, $-_S$, $=_S$, $\leq_S$, $\neg_T$ and $\wedge_T$ are defined in Tables 2.4 and 2.6. Furthermore, we have

$$\mathcal{FZ}[\![n]\!]\ ps = \mathsf{abs}_Z(\mathcal{N}[\![n]\!])$$

$$\mathcal{FT}[\![\mathtt{true}]\!]\ ps = \mathrm{TT} \text{ and } \mathcal{FT}[\![\mathtt{false}]\!]\ ps = \mathrm{FF}$$

$$\mathcal{FS}[\![x := a]\!]\ ps = ps[x \mapsto \mathcal{FA}[\![a]\!]ps]$$

$$\mathcal{FS}[\![\texttt{skip}]\!] = \mathsf{id}$$

$$\mathcal{FS}[\![S_1;S_2]\!] = \mathcal{FS}[\![S_2]\!] \circ \mathcal{FS}[\![S_1]\!]$$

$$\mathcal{FS}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_s]\!] = \mathsf{cond}_F(\mathcal{FB}[\![b]\!], \mathcal{FS}[\![S_1]\!], \mathcal{FS}[\![S_2]\!])$$

$$\mathcal{FS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \mathsf{FIX}\ H$$
$$\text{where } H\ h = \mathsf{cond}_F(\mathcal{FB}[\![b]\!], h \circ \mathcal{FS}[\![S]\!], \mathsf{id})$$

Table 4.2: Forward analysis of statements

so the property states are ignored when determining the properties of numbers and truth values.

The forward analysis of a statement will be defined by a function $\mathcal{FS}$ of functionality:

$$\mathcal{FS}\text{: } \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

The idea is that if ps is a property of the initial state of S then $\mathcal{FS}[\![S]\!]$ ps is a property of the final state obtained by executing S from the initial state. The totality of $\mathcal{FS}[\![S]\!]$ reflects that we shall be able to analyse *all* statements of **While** including statements that loop in the direct style semantics. The definition of $\mathcal{FS}$ is given by the clauses of Table 4.2 and they are parameterised on the definition of cond_F:

$$\textbf{F10:} \qquad \mathsf{cond}_F\text{: } ((\mathbf{PState} \to \mathbf{P}_T) \times (\mathbf{PState} \to \mathbf{PState}) \times (\mathbf{PState} \to \mathbf{PState})) \to (\mathbf{PState} \to \mathbf{PState})$$

specifying how to analyse the conditional.

Example 4.1.3 For the detection of signs analysis we have

$$\mathsf{cond}_F(f, h_1, h_2)ps = \begin{cases} \text{INIT} & \text{if } f\ ps = \text{NONE} \\ h_1\ ps & \text{if } f\ ps = \text{TT} \\ h_2\ ps & \text{if } f\ ps = \text{FF} \\ (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \text{ANY} \end{cases}$$

In summary, to specify a forward program analysis of **While** we only have to provide definitions of the lattices of **P1** and **P2** and to define the functions of **F1** – **F10**.

$$
\begin{array}{lcl}
\mathcal{BA}[\![n]\!]\ p & = & \mathcal{BZ}[\![n]\!]\ p \\
\mathcal{BA}[\![x]\!]\ p & = & \textsc{init}[x \mapsto p] \\
\mathcal{BA}[\![a_1 + a_2]\!]\ p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{add}_B\ p) \\
\mathcal{BA}[\![a_1 \star a_2]\!]\ p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{mult}_B\ p) \\
\mathcal{BA}[\![a_1 - a_2]\!]\ p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{sub}_B\ p) \\
\\
\mathcal{BB}[\![\mathtt{true}]\!]\ p & = & \mathcal{BT}[\![\mathtt{true}]\!]\ p \\
\mathcal{BB}[\![\mathtt{false}]\!]\ p & = & \mathcal{BT}[\![\mathtt{false}]\!]\ p \\
\mathcal{BB}[\![a_1 = a_2]\!]\ p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{eq}_B\ p) \\
\mathcal{BB}[\![a_1 \leq a_2]\!]\ p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{leq}_B\ p) \\
\mathcal{BB}[\![\neg\ b]\!]\ p & = & \mathcal{BB}[\![b]\!](\mathsf{neg}_B\ p) \\
\mathcal{BB}[\![b_1\ \wedge\ b_2]\!]\ p & = & \mathsf{join}(\mathcal{BB}[\![b_1]\!], \mathcal{BB}[\![b_2]\!])(\mathsf{and}_B\ p)
\end{array}
$$

Table 4.3: Backward analysis of expressions

Backward Analysis

In a backward analysis the computation is performed in the *opposite direction* of the direct style semantics: given properties of the output of the computation the analysis will predict the properties the input should have. Thus the idea will be to replace the semantics functions

$$\mathcal{A}\text{: } \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Z}$$

$$\mathcal{B}\text{: } \mathbf{Bexp} \to \mathbf{State} \to \mathbf{T}$$

$$\mathcal{S}_{ds}\text{: } \mathbf{Stm} \to \mathbf{State} \hookrightarrow \mathbf{State}$$

with semantic functions that not only compute with properties rather than values but also "invert the function arrows":

$$\mathcal{BA}\text{: } \mathbf{Aexp} \to \mathbf{P}_Z \to \mathbf{PState}$$

$$\mathcal{BB}\text{: } \mathbf{Bexp} \to \mathbf{P}_T \to \mathbf{PState}$$

$$\mathcal{BS}\text{: } \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

For an arithmetic expression a the idea is that given a property p of the result of computing a, $\mathcal{BA}[\![a]\!]\ p$ will be a property state telling which properties the variables of a should have in order for the result of computing a to have property p. So for the result of evaluating a variable x to have property p we simply assume that the variable has that property and we have no assumptions about the other variables. As another example consider the analysis of the expression $a_1 + a_2$ and assume that the result of evaluating it should be p; we will then determine which properties the variables of a_1+a_2 should have in order for this to be the case. The first step of the analysis will be to determine which properties the result of evaluating the subexpressions a_1 and a_2 should have in order for the result of $a_1 + a_2$ to be p. Let that be p_1 and p_2. We can now analyse the subexpressions: the analysis of a_i from p_i will find out which properties the variables of a_i should have initially in order for the result of a_i to have property p_i. The last step will be to combine the information from the two subexpressions and this will often be a least upper bound operation. Thus the analysis can be specified as

$$\mathcal{BA}[\![a_1 + a_2]\!]\ p = \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])\ (\mathsf{add}_B\ p)$$

where

$$\mathsf{join}(h_1, h_2)(p_1, p_2) = (h_1\ p_2)\ \sqcup_{PS}\ (h_2\ p_2)$$

and $\sqcup_{PS}$ is the least upper bound on property states.

Similar remarks hold for the backward analysis of booleans expressions. The clauses are given in Table 4.3 and as was the case for the forward analysis they are parameterised on the auxiliary functions specifying how the arithmetic and boolean operators should be analysed:

B1:	add_B: $\mathbf{P}_Z \to \mathbf{P}_Z \times \mathbf{P}_Z$
B2:	mult_B: $\mathbf{P}_Z \to \mathbf{P}_Z \times \mathbf{P}_Z$
B3:	sub_B: $\mathbf{P}_Z \to \mathbf{P}_Z \times \mathbf{P}_Z$
B4:	eq_B: $\mathbf{P}_T \to \mathbf{P}_Z \times \mathbf{P}_Z$
B5:	leq_B: $\mathbf{P}_T \to \mathbf{P}_Z \times \mathbf{P}_Z$
B6:	neg_B: $\mathbf{P}_T \to \mathbf{P}_T$
B7:	and_B: $\mathbf{P}_T \to \mathbf{P}_T \times \mathbf{P}_T$

We need a way of turning numbers and truth values into property states. Given a number n and a property p the analysis $\mathcal{BA}$ has to determine a property state that will ensure that the result of evaluating n will have property p. However, it might be the case that n cannot have the property p at all and in this case the property state returned by $\mathcal{BA}[\![n]\!]\ p$ should differ from that returned if n can have the property p. Therefore we shall assume that we have functions

$$
\begin{aligned}
&\mathcal{BS}[\![x := a]\!]\ ps = \mathsf{update}_B(ps,\ x,\ \mathcal{BA}[\![a]\!])\\
&\mathcal{BS}[\![\mathtt{skip}]\!] = \mathsf{id}\\
&\mathcal{BS}[\![S_1;S_2]\!] = \mathcal{BS}[\![S_1]\!] \circ \mathcal{BS}[\![S_2]\!]\\
&\mathcal{BS}[\![\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_s]\!] = \mathsf{cond}_B(\mathcal{BB}[\![b]\!],\ \mathcal{BS}[\![S_1]\!],\ \mathcal{BS}[\![S_2]\!])\\
&\mathcal{BS}[\![\mathtt{while}\ b\ \mathtt{do}\ S]\!] = \mathsf{FIX}\ H\\
&\qquad \text{where } H\ h = \mathsf{cond}_B(\mathcal{BB}[\![b]\!],\ \mathcal{BS}[\![S]\!] \circ h,\ \mathsf{id})
\end{aligned}
$$

Table 4.4: Backward analysis of statements

B8: $\mathcal{BZ}$: $\mathbf{Num} \to \mathbf{P}_Z \to \mathbf{PState}$

B9: $\mathcal{BT}$: $\{\mathtt{true}, \mathtt{false}\} \to \mathbf{P}_T \to \mathbf{PState}$

that take the actual property into account when turning numbers and truth values into property states.

Turning to statements we shall specify their analysis by a function $\mathcal{BS}$ of functionality:

$$\mathcal{BS}\colon \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

Again the totality of $\mathcal{BS}[\![S]\!]$ reflects that we shall be able to analyse *all* statements including those that loop in the direct style semantics. Since $\mathcal{BS}$ is a backward analysis the argument ps of $\mathcal{BS}[\![S]\!]$ will be a property state corresponding to the result of executing S and $\mathcal{BS}[\![S]\!]\ ps$ will be the property state corresponding to the initial state from which S is executed. The definition of $\mathcal{BS}$ is given in Table 4.4. We shall assume that we have a functions

B10: update_B: $\mathbf{PState} \times \mathbf{Var} \times (\mathbf{P}_Z \to \mathbf{PState}) \to \mathbf{PState}$

B11: cond_B: $((\mathbf{P}_T \to \mathbf{PState}) \times (\mathbf{PState} \to \mathbf{PState}) \times (\mathbf{PState} \to \mathbf{PState})) \to (\mathbf{PState} \to \mathbf{PState})$

specifying how to analyse the assignment and the conditional.

4.2 Dependency Analysis

The detection of signs analysis is an example of a *forward first order analysis*: we are concerned with properties of the values arising during computation. We shall now present a *forward second order analysis*: in a *dependency analysis* we are concerned with properties of the relationship between values. The properties we shall study are quite simple: EQ for when two values are equal and ANY for when two values might be unequal.

The goal of the dependency analysis will be to regard some program variables as *input* variables and some as *output* variables and the analysis will then be used to determine whether or not the final values of the output variables only depend on the initial values of the input variables. If so, we shall say that there is a *functional dependency* between the input and output variables of the program.

As an example, in the factorial program

```
y := 1; while ¬(x = 1) do (y := y ⋆ x; x := x − 1)
```

we will think of `x` as an input variable and `y` as an output variable. We may then ask whether the final values of the output variables *only* depend upon the initial values of the input variables. This is clearly the case for the program above but it is not the case for all programs. An example is

```
while ¬(x = 1) do (y := y ⋆ x; x := x − 1)
```

where we still assume that `x` is the input variable and `y` is the output variable. Here the variable `y` is uninitialized and therefore its value in the final state does not only depend on the value of the input variable `x` but also on the value of `y` in the initial state.

The *specification* of the analysis falls into two parts. First we introduce the properties that the analysis operates on and next we specify the actual analysis for the three syntactic categories.

Properties and property states

For the dependency analysis we shall be interested in two properties of the relationship between values (numbers or truth values):

- EQ meaning that the values *definitely* are equal, and
- ANY meaning that the values *may* not be equal.

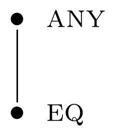
We shall write

$$\mathbf{P} = \{\text{EQ}, \text{ANY}\}$$

for this set of properties and we use p as a meta-variable ranging over $\mathbf{P}$. It is more informative to know that an expression has the property EQ than ANY. As a record of this we define a partial order $\sqsubseteq_P$ on $\mathbf{P}$:

$$\text{EQ} \sqsubseteq_P \text{ ANY}, \quad \text{EQ} \sqsubseteq_P \text{ EQ}, \quad \text{ANY} \sqsubseteq_P \text{ ANY}$$

which may be depicted as



Thus the more informative property is at the bottom of the ordering.

Fact 4.1 $(\mathbf{P}, \sqsubseteq_P)$ is a complete lattice. If Y is a subset of $\mathbf{P}$ then

$\bigsqcup_P Y = \text{ANY}$ if and only if $\text{ANY} \in Y$

Given *two* values we can associate a "best" property with them. In the case of numbers we define the function

$$\mathsf{abs}_Z\colon \mathbf{Z} \times \mathbf{Z} \to \mathbf{P}$$

defined by

$$\mathsf{abs}_Z\ (z_1, z_2) = \begin{cases} \text{EQ} & \text{if } z_1 = z_2 \\ \text{ANY} & \text{otherwise} \end{cases}$$

In the case of truth values we define

$$\mathsf{abs}_T\colon \mathbf{T} \times \mathbf{T} \to \mathbf{P}$$

defined by

$$\mathsf{abs}_T\ (t_1, t_2) = \begin{cases} \text{EQ} & \text{if } t_1 = t_2 \\ \text{ANY} & \text{otherwise} \end{cases}$$

Following the approach of the previous analysis we introduce a *property state* mapping variables to properties. The property states will express properties of relationships between states. It turns out that it does not make sense to compare states that are "too different": they must have comparable "histories". To capture this we introduce a special token $\mathsf{history}$ that captures the "flow of control"; it acts like an "extended program counter". The set **PState** of property states ranged over by the meta-variable ps, is then defined by

$$\mathbf{PState} = (\mathbf{Var} \ \cup \ \{\mathsf{history}\}) \rightarrow \mathbf{P}$$

The idea is that if $\mathsf{history}$ is mapped to $\textsc{eq}$ then the two states have the same "history"; if it is mapped to $\textsc{any}$ this need not be the case. For a property state $ps \ \in \ \mathbf{PState}$ we define the set

$$\mathrm{EQ}(ps) = \{ \ x \ \in \ \mathbf{Var} \ \cup \ \{\mathsf{history}\} \mid ps \ x = \textsc{eq} \ \}$$

of "variables" mapped to $\textsc{eq}$ and we say that

ps is *proper* if and only if $ps(\mathsf{history}) = \textsc{eq}$.

If ps is not proper we shall sometimes say that it is *improper*.

The operation abs_Z may be lifted to states; for this we define

$$\mathsf{abs}\text{: } \mathbf{State} \ \times \ \mathbf{State} \rightarrow \mathbf{PState}$$

in a component-wise manner:

$$\mathsf{abs}(s_1, s_2) \ x = \mathsf{abs}_Z(s_1 \ x, s_2 \ x)$$
$$\mathsf{abs}(s_1, s_2) \ \mathsf{history} = \textsc{eq}.$$

Our next task will be to endow **PState** with some partially ordered structure: In Lemma 1.1 we instantiate S to be $\mathbf{Var} \ \cup \ \{\mathsf{history}\}$ and D to be $\mathbf{P}$ and we get:

Corollary 4.2 Let $\sqsubseteq_{PS}$ be the ordering on **PState** defined by

$$ps_1 \sqsubseteq_{PS} ps_2$$

if and only if

$$ps_1 \ x \sqsubseteq_P \ ps_2 \ x \text{ for all } x \ \in \ \mathbf{Var} \ \cup \ \{\mathsf{history}\}$$

Then (**PState**, $\sqsubseteq_{PS}$) is a complete lattice. In particular, the least upper bound $\bigsqcup_{PS} Y$ of a subset Y of **PState** is characterized by

$$(\bigsqcup_{PS} Y)\ x = \bigsqcup_{PS} \{\ ps\ x \mid ps\ \in\ \ Y\ \}$$

We shall write FAIL for the property state ps that maps all variables to ANY and that maps history to ANY. Similarly, we shall write INIT for the property state that maps all variables to EQ and that maps history to EQ. Note that INIT is the *least element* of **PState**.

Example 4.3 To motivate the need for keeping track of the history, that is the need for history, consider the following statement

S: `if x = 1 then x := 1 else x := 2`

Assume first that we do *not* keep track of the flow of control. The analysis of each of the two branches will give rise to a function that maps ps to $ps[\mathtt{x}\mapsto\text{EQ}]$ so it would be natural to expect the analysis of S to do the same. However, this will *not always be correct*. To see this suppose that ps, s_1 and s_2 are such that

$ps\ \mathtt{x} = \text{ANY}$ and $ps\ y = \text{EQ}$ otherwise

$s_1\ \mathtt{x} = \mathbf{1}$ and $s_1\ y = \mathbf{0}$ otherwise

$s_2\ \mathtt{x} = \mathbf{2}$ and $s_2\ y = \mathbf{0}$ otherwise

Then clearly

$\mathsf{abs}(s_1,\ s_2) \sqsubseteq_{PS} ps$

so ps captures the relationship between the two initial states. But it is *not* the case that $\mathsf{abs}(\mathcal{S}_{ds}[\![S]\!]s_1,\ \mathcal{S}_{ds}[\![S]\!]s_2) \sqsubseteq_{PS} ps[\mathtt{x}\mapsto\text{EQ}]$ because $\mathcal{S}_{ds}[\![S]\!]s_1 = s_1$ and $\mathcal{S}_{ds}[\![S]\!]s_2 = s_2$ and $s_1\ \mathtt{x} \neq s_2\ \mathtt{x}$. Thus the result $ps[\mathtt{x}\mapsto\text{EQ}]$ of the analysis does *not* capture the relationship between the final states – a property we definitely want the analysis to have in order to express its safety.

The token history is used to solve this dilemma. The key observation is that if $ps\ \mathtt{x} = \text{ANY}$ then the outcome of the test `x = 1` may differ for different states described by ps. Thus the flow of control does not need to be the same for the two states. This is captured by the token history: if the analysis of the test gives ANY then history will be mapped to ANY in the resulting property state and otherwise it is mapped to EQ. We shall return to this example in more detail when we have specified the analysis for statements. □

Exercise 4.4 Show that

$$ps_1 \sqsubseteq_{PS} ps_2 \text{ if and only if } \mathrm{EQ}(ps_1) \supseteq \mathrm{EQ}(ps_2\,)$$

Next show that

$$\mathrm{EQ}(\bigsqcup_{PS} Y) = \bigcap \{\ \mathrm{EQ}(ps) \mid ps\ \in\ Y\ \}$$

whenever Y is a non-empty subset of **PState**. □

Analysis of expressions

The analysis of an arithmetic expression a will be specified by a (total) function $\mathcal{DA}[\![a]\!]$ from property states to properties:

$$\mathcal{DA}\text{: } \mathbf{Aexp} \to \mathbf{PState} \to \mathbf{P}$$

Similarly, the analysis of a boolean expression b will be defined by a (total) function $\mathcal{DB}[\![b]\!]$ from property states to properties:

$$\mathcal{DB}\text{: } \mathbf{Bexp} \to \mathbf{PState} \to \mathbf{P}$$

The defining clauses are given in Table 4.5.

The overall idea is that once ps history has the value ANY then all results produced should be ANY. This is reflected directly in the clauses for the basic constructs n, x, `true` and `false`. For the composite expression, as for example $a_1 + a_2$, the idea is that it can only have the property EQ if both subexpressions have that property. This is ensured by the binary operation $\sqcup_P$.

The functions $\mathcal{DA}[\![a]\!]$ and $\mathcal{DB}[\![b]\!]$ are closely connected with the sets of free variables defined in Chapter 1 of [NN]:

Exercise 4.5 Prove that for every arithmetic expression a we have

$$\mathcal{DA}[\![a]\!]ps = \textsc{eq} \text{ if and only if } \mathrm{FV}(a)\ \cup\ \{\mathsf{history}\} \subseteq \mathrm{EQ}(ps)$$

Formulate and prove a similar result for boolean expressions. Deduce that for all a of **Aexp** we get $\mathcal{DA}[\![a]\!]ps = \textsc{any}$ if ps is improper, and that for all b of **Bexp** we get $\mathcal{DB}[\![b]\!]ps = \textsc{any}$ if ps is improper. □

$$
\begin{array}{lcl}
\mathcal{DA}[\![n]\!]ps & = & \begin{cases} \text{EQ} & \text{if } ps \ \mathsf{history} = \text{EQ} \\ \text{ANY} & \text{otherwise} \end{cases} \\
\mathcal{DA}[\![x]\!]ps & = & \begin{cases} ps\ x & \text{if } ps \ \mathsf{history} = \text{EQ} \\ \text{ANY} & \text{otherwise} \end{cases} \\
\mathcal{DA}[\![a_1 + a_2]\!]ps & = & (\mathcal{DA}[\![a_1]\!]ps) \sqcup_P (\mathcal{DA}[\![a_2]\!]ps) \\
\mathcal{DA}[\![a_1 \star a_2]\!]ps & = & (\mathcal{DA}[\![a_1]\!]ps) \sqcup_P (\mathcal{DA}[\![a_2]\!]ps) \\
\mathcal{DA}[\![a_1 - a_2]\!]ps & = & (\mathcal{DA}[\![a_1]\!]ps) \sqcup_P (\mathcal{DA}[\![a_2]\!]ps) \\
\mathcal{DB}[\![\texttt{true}]\!]ps & = & \begin{cases} \text{EQ} & \text{if } ps \ \mathsf{history} = \text{EQ} \\ \text{ANY} & \text{otherwise} \end{cases} \\
\mathcal{DB}[\![\texttt{false}]\!]ps & = & \begin{cases} \text{EQ} & \text{if } ps \ \mathsf{history} = \text{EQ} \\ \text{ANY} & \text{otherwise} \end{cases} \\
\mathcal{DB}[\![a_1 = a_2]\!]ps & = & (\mathcal{DA}[\![a_1]\!]ps) \sqcup_P (\mathcal{DA}[\![a_2]\!]ps) \\
\mathcal{DB}[\![a_1 \leq a_2]\!]ps & = & (\mathcal{DA}[\![a_1]\!]ps) \sqcup_P (\mathcal{DA}[\![a_2]\!]ps) \\
\mathcal{DB}[\![\neg\, b]\!]ps & = & \mathcal{DB}[\![b]\!]ps \\
\mathcal{DB}[\![b_1 \wedge b_2]\!]ps & = & (\mathcal{DB}[\![b_1]\!]ps) \sqcup_P (\mathcal{DB}[\![b_2]\!]ps)
\end{array}
$$

Table 4.5: Dependency analysis of expressions

Analysis of statements

Turning to statements we shall specify their analysis by a function $\mathcal{DS}$ of functionality:

$$\mathcal{DS}\colon \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

The totality of $\mathcal{DS}[\![S]\!]$ reflects that we shall be able to analyse *all* statements including a statement like `while true do skip` that loops. The definition of $\mathcal{DS}$ is given in Table 4.6.

The clauses for assignment, `skip` and composition are much as in the direct style denotational semantics of Chapter 4 of [NN]. In the clause for `if` b `then` S_1 `else` S_2 we use the auxiliary function cond_D defined by

$$\mathsf{cond}_D(f,\, h_1,\, h_2)\ ps = \begin{cases} (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \text{EQ} \\ \text{FAIL} & \text{if } f\ ps = \text{ANY} \end{cases}$$

$$\mathcal{DS}[\![x := a]\!]\ ps = ps[x \mapsto \mathcal{DA}[\![a]\!]ps]$$

$$\mathcal{DS}[\![\texttt{skip}]\!] = \mathsf{id}$$

$$\mathcal{DS}[\![S_1;S_2]\!] = \mathcal{DS}[\![S_2]\!] \circ \mathcal{DS}[\![S_1]\!]$$

$$\mathcal{DS}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_s]\!] = \mathsf{cond}_D(\mathcal{DB}[\![b]\!], \mathcal{DS}[\![S_1]\!], \mathcal{DS}[\![S_2]\!])$$

$$\mathcal{DS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \mathsf{FIX}\ H$$
$$\text{where } H\ h = \mathsf{cond}_D(\mathcal{DB}[\![b]\!], h \circ \mathcal{DS}[\![S]\!], \mathsf{id})$$

Table 4.6: Dependency analysis of statements

First consider the case where we are successful in analysing the condition, that is where $f\ ps = \text{EQ}$. For each variable x we can determine the result of analysing each of the branches, namely $(h_1\ ps)\ x$ for the true branch and $(h_2\ ps)\ x$ for the false branch. The least upper bound of these two results will be the new property bound to x, that is the new property state will map x to

$$((h_1\ ps)\ x) \sqcup_P ((h_2\ ps)\ x)$$

If the analysis of the condition is not successful, that is $f\ ps = \text{ANY}$, then the analysis of the conditional will fail and we shall therefore use the property state FAIL.

Example 4.6 Returning to Example 4.3 we see that if $ps\ \texttt{x} = \text{ANY}$ then

$$\mathcal{DB}[\![\texttt{x = 1}]\!]ps = \text{ANY}$$

We therefore get

$$\mathcal{DS}[\![\texttt{if x = 1 then x := 1 else x := 2}]\!]ps = \text{FAIL}$$

using the above definition of cond_D. □

Example 4.7 Consider now the statement

```
if x = x then x := 1 else x := y
```

First assume that $ps\ \texttt{x} = \text{EQ}$, $ps\ \texttt{y} = \text{ANY}$ and $ps\ \mathsf{history} = \text{EQ}$. Then $\mathcal{DA}[\![\texttt{x = x}]\!]\ ps = \text{EQ}$ and we get

$$\begin{aligned}
&\mathcal{DS}[\![\texttt{if x = x then x := 1 else x := y}]\!]\ ps\ \texttt{x}\\
&\quad = \mathsf{cond}_D(\mathcal{DB}[\![\texttt{x = x}]\!], \mathcal{DS}[\![\texttt{x := 1}]\!], \mathcal{DS}[\![\texttt{x := y}]\!])\ ps\ \texttt{x}\\
&\quad = (\mathcal{DS}[\![\texttt{x := 1}]\!]\ ps \sqcup_{PS} \mathcal{DS}[\![\texttt{x := y}]\!]\ ps)\ \texttt{x}\\
&\quad = \text{ANY}
\end{aligned}$$

because $\mathcal{DB}[\![\texttt{x = x}]\!]ps = \text{EQ}$, $(\mathcal{DS}[\![\texttt{x := 1}]\!]ps)\ \texttt{x} = \text{EQ}$ but $(\mathcal{DS}[\![\texttt{x := y}]\!]ps)$ $\texttt{x} = \text{ANY}$. So even though the false branch never will be executed it will influence the result obtained by the analysis.

Next assume that $ps\ \texttt{x} = \text{ANY}$, $ps\ \texttt{y} = \text{EQ}$ and $ps\ \mathsf{history} = \text{EQ}$. Then $\mathcal{DA}[\![\texttt{x = x}]\!]\ ps = \text{ANY}$ and we get

$$\begin{aligned}
&\mathcal{DS}[\![\texttt{if x = x then x := 1 else x := y}]\!]\ ps\\
&\quad = \mathsf{cond}_D(\mathcal{DB}[\![\texttt{x = x}]\!], \mathcal{DS}[\![\texttt{x := 1}]\!], \mathcal{DS}[\![\texttt{x := y}]\!])\ ps\\
&\quad = \text{FAIL}
\end{aligned}$$

because $\mathcal{DB}[\![\texttt{x = x}]\!]ps = \text{ANY}$. So even though the test always evaluates to true for all states, this is not captured by the analysis. More complex analyses could do better (for example by trying to predict the outcome of tests). □

In the clause for the `while`-loop we also use the function cond_D and otherwise the clause is as in the direct style denotational semantics of Chapter 4 of [NN]. In particular we use the fixed point operation FIX as it corresponds to unfolding the `while`-loop any number of times. As in Chapter 4 of [NN] the fixed point is defined by

$$\mathsf{FIX}\ H = \bigsqcup \{\ H^n \perp \mid n \ \geq\ 0\ \}$$

where the functionality of H is

$$H\colon (\textbf{PState} \to \textbf{PState}) \to (\textbf{PState} \to \textbf{PState})$$

and where **PState** → **PState** is the set of total functions from **PState** to **PState**. In order for this to make sense H must be a continuous function on a ccpo with least element $\perp$.

Example 4.8 We are now in a position where we can attempt the application of the analysis to the factorial statement:

$$\mathcal{DS}[\![\texttt{y:=1; while}\ \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x}-\texttt{1)}]\!]$$

We shall apply this function to the *proper* property state ps_0 that maps `x` to EQ and all other variables (including `y`) to ANY as this corresponds to viewing `x` as the only input variable of the statement.

To do so we use the clauses of Tables 4.5 and 4.6 and get

$$\begin{aligned}&\mathcal{DS}[\![\texttt{y:=1; while }\neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x}-\texttt{1)}]\!]\ ps_0\\&\qquad = (\textsf{FIX}\ H)\ (ps_0[\texttt{y}\mapsto\text{EQ}])\end{aligned}$$

where

$$H\ h = \textsf{cond}_D(\mathcal{DB}[\![\neg\texttt{(x=1)}]\!], h \circ \mathcal{DS}[\![\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}]\!], \text{id})$$

We first simplify H and obtain

$$(H\ h)\ ps = \begin{cases}\text{FAIL} & \text{if } ps\ \textsf{history} = \text{ANY or } ps\ \texttt{x} = \text{ANY}\\ (h\ ps) \sqcup_{PS} ps & \text{if } ps\ \textsf{history} = \text{EQ and } ps\ \texttt{x} = \text{EQ}\end{cases}$$

At this point we shall exploit the result of Exercise 2.7:

$$\begin{aligned}&\text{if } H^n\ \bot = H^{n+1}\ \bot \text{ for some } n\\&\text{then } \textsf{FIX}\ H = H^n\ \bot\end{aligned}$$

where $\bot$ is the function $\bot\ ps$ = INIT for all ps. We can now calculate the iterands $H^1\ \bot$, $H^1\ \bot$, $\cdots$. We obtain

$$\begin{aligned}(H^0\ \bot)\ ps &= \text{INIT}\\ (H^1\ \bot)\ ps &= \begin{cases}\text{FAIL} & \text{if } ps\ \texttt{x} = \text{ANY or } ps \text{ not proper}\\ ps & \text{if } ps\ \texttt{x} = \text{EQ and } ps \text{ proper}\end{cases}\\ (H^2\ \bot)\ ps &= \begin{cases}\text{FAIL} & \text{if } ps\ \texttt{x} = \text{ANY or } ps \text{ not proper}\\ ps & \text{if } ps\ \texttt{x} = \text{EQ and } ps \text{ proper}\end{cases}\end{aligned}$$

where ps is an arbitrary property state. Since $H^1\ \bot = H^2\ \bot$ our assumption above ensures that we have found the least fixed point for H:

$$(\textsf{FIX}\ H)\ ps = \begin{cases}\text{FAIL} & \text{if } ps\ \texttt{x} = \text{ANY or } ps \text{ not proper}\\ ps & \text{if } ps\ \texttt{x} = \text{EQ and } ps \text{ proper}\end{cases}$$

It is now straightforward to verify that $(\textsf{FIX}\ H)\ (ps_0[\texttt{y}\mapsto\text{EQ}])\ \texttt{y} = \text{EQ}$ and that $(\textsf{FIX}\ H)(ps_0[\texttt{y}\mapsto\text{EQ}])$ is proper. We conclude that there *is* a functional dependency between the input variable `x` and the output variable `y`. □

Exercise 4.9 Extend **While** with the statement `repeat` S `until` b and give the new (compositional) clause for $\mathcal{DS}$. Motivate your extension. □

Exercise 4.10 Prove the existence of the dependency analysis as specified by $\mathcal{DA}$, $\mathcal{DB}$ and $\mathcal{DS}$. □

Exercise 4.11 Define the function

$$\mathsf{prop}\colon \mathbf{PState} \to \mathbf{PState}$$

by

$$\mathsf{prop}\ ps\ x = (ps\ x) \sqcup_P (ps\ \mathsf{history})$$

and note that $\mathsf{prop}\ ps = ps$ when ps is proper and $\mathsf{prop}\ ps = \textsc{fail}$ otherwise. Define the safety predicate

$$g\ \mathsf{safe}_A\ h$$

by

$$\begin{array}{l} \mathsf{abs}(s_1,\ s_2) \sqsubseteq_{PS} \mathsf{prop}\ ps \\ \Downarrow \\ \mathsf{abs}_Z(g\ s_1,\ g\ s_2) \sqsubseteq_P h\ ps \end{array}$$

Prove that $\mathcal{A}[\![a]\!]\ \mathsf{safe}_A\ \mathcal{DA}[\![a]\!]$ holds for all arithmetic expressions $a \in \mathbf{Aexp}$.

Perform a similar development for the analysis of boolean expressions. □

Exercise 4.12 Define the safety predicate

$$g\ \mathsf{safe}_S\ h$$

by

$$\begin{array}{l} \mathsf{abs}(s_1,\ s_2) \sqsubseteq_{PS} \text{prop}\ ps \\ \Downarrow \\ (g\ s_1 \neq \underline{\text{undef}} \wedge g\ s_2 \neq \underline{\text{undef}})\ \text{implies} \\ \qquad\qquad \mathsf{abs}(g\ s_1,\ g\ s_2) \sqsubseteq_{PS} \mathsf{prop}\ (h\ ps),\ \text{and} \\ h\ ps\ \text{proper implies} \\ \qquad\qquad (g\ s_1 \neq \underline{\text{undef}}\ \text{if and only if}\ g\ s_2 \neq \underline{\text{undef}}) \end{array}$$

Prove that $\mathcal{S}_{ds}[\![S]\!]$ safe$_S$ $\mathcal{DS}[\![S]\!]$ holds for all statements $S \in$ **Stm**. □

Exercise 4.13 One may consider replacing the definition of cond_D by

$$\mathsf{cond}'_D(f, h_1, h_2)ps = (h_1\ ps) \sqcup_{PS} (h_2\ ps)$$

Formally show that the resulting analysis cannot be correct in the sense of Exercise 4.12. □

Exercise 4.14 In the above exercise we saw that cond_D could not be simplified so as to ignore the test. Now consider the following remedy

$$\mathsf{cond}'_D(f, h_1, h_2)ps = \begin{cases} (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \mathrm{EQ} \\ ((h_1\ ps') \sqcup_{PS} (h_2\ ps'))[\mathsf{history} \mapsto \mathrm{EQ}] & \\ \qquad \text{if } f\ ps = \mathrm{ANY} \text{ and } ps' = ps[\mathsf{history} \mapsto \mathrm{ANY}] & \end{cases}$$

Give an example statement where cond'_D is preferable to cond_D. □

Exercise 4.15 The results developed in Chapter 3 for computing fixed points are applicable to the functional dependency analysis:

- show that the analysis is in the completely additive framework, that is, that the functions $\mathcal{DS}[\![S]\!]$ are strict and additive for all statements S of **While**, and
- show that the functionals H obtained for `while`-loops can be written in iterative form.

Conclude, using Theorem 3.10 that in a program with p variables at most $p + 1$ iterations are needed to compute the fixed point. □