MAC ISO Term 2 - “Abstract Interpretation”

David P. Cunningham
Supervisor: Herbert Wiklicky

April 25, 2005

Abstract

A summary of the motivation and theory behind abstract interpretation, including the accumulating semantics, Galois connections and widening. A complete demonstration of the use of abstract interpretation to define a safe and optimal sign analysis in the context of a simple imperative language is presented. In addition, a example of widening is described to improve the termination properties of an interval analysis of the same language.

Keywords:

- Semantics
- Program analysis
- Abstract interpretation
- Widening

Contents

Chapter 1

Introduction and Preliminaries

Computer Science has long been interested in the potential for computers to understand the behaviour of the programs we write, in spite of serious theoretical problems which make all but the most simple "analyses" impractical. These simple analyses, although incomplete, are still found in many commonly used compilers. They collect information that supports error detection as well as various kinds of optimisations. Abstract interpretation is the theory that connects the semantics of a programming language (the precise behaviour attributed to the language by its designers), to a cut-down model of the language that a computer can realistically understand.

An "understanding" of a program typically means the assurance of certain properties, e.g. if a variable is always positive at a certain point, or if a variable is ever used. We will see how abstract interpretation can be used to derive an appropriate analysis from a basic specification. The specification takes the form of a mapping from the concrete and undecidable world of *ultimately precise* program properties, to some decidable *restriction* of this information. The derived analysis takes the form of an algorithm that can compute the restricted properties without error.

The theory was pioneered by Cousot and Cousot [2] [3] [4]. This report also draws from the article [6] and parts of [7].

In order to formalise an approximate program analysis, we need several preliminaries: We must rigorously define the semantics of the language in which our programs are written. We translate this into an ultimately precise analysis (with which all program properties can be discovered). This is the starting point for all our derived analyses. It is also helpful to give an example of a derived, approximate, analysis, initially without any proof of correctness. This demonstrates what the method of abstract interpretation will provide. The remainder of this chapter is therefore an account of the formalisation of these preliminary concepts and ideas.

1.1 An example language

We can consider the set of all possible programs by describing the language in which one can write the programs. For the purposes of explaining and demonstrating abstract interpretation, we use a simple imperative language very similar to the language "while" used in [1]. The language supports tests, loops, arithmetic evaluation and assignments. Programs are very simple and thus also simple to analyse, making the demonstration of abstract interpretation much easier.

We choose to use flow charts instead of conventional abstract syntax, since although it is possible to consider languages in terms of their abstract syntax (as in [1]) this is not relevant to the theory of abstract interpretation. The flow chart representation can be derived from a given program's abstract syntax using a simple algorithm that is not given here (this is essentially the approach in [1]). We instead state our example programs directly in the flow chart representation which is based on that used in [2], [3], and [4].

Definition 1.1.1. A program is a directed graph: a set of nodes and a set of edges (arcs). Nodes can be either *assignments*, *tests*, or *joins*.

Definition 1.1.2. The set of *arcs* is denoted **Arc** and the symbol r will range over arcs.

The *type* of node places limits on the number of arcs that are allowed to link to and from that node.

- *Assignments* have one input and one output arc.
- *Tests* have one input and two output arcs.
- *Joins* have more than one input arc and one output arc.

Sometimes arcs will be only connected to a single node, either having no predecessor or no successor. Such arcs will hereafter be called "entry" and "exit" arcs respectively. In keeping with pragmatic programming languages, exactly one entry arc is allowed in the program, but also multiple exit arcs are not allowed since it is possible to join together proposed exit arcs with a join node. We also require the graph to be connected, since any parts disconnected from the entry arc will have no effect on the computation.

Graphically, the assignment, test and join nodes are represented with rectangles, diamonds and filled circles respectively. Assignment nodes contain a text label that defines exactly what assignment is being performed, and similarly for test nodes. The two exits from a test node are distinctly labelled either `T` or `F`. For clarity, entry arcs are shown to point from a place-holder triangle symbol (multimedia "play"). Likewise for exit arcs point to a square symbol (multimedia "stop"). Thus there is no ambiguity when we consider "partial" programs where empty successor / predecessor indicates that the rest of the program has not been drawn.

Since our language is imperative, we must consider the "environment" - a mutable variable store. This is represented by a function from the finite set of variables used in the program to a set of values. We *could* use other representations here, but we can construct maps very succinctly with lambda notation.

Definition 1.1.3. The finite set of *variables* used in the program is denoted by **Var**, and x will range over variables.

Definition 1.1.4. The set of *values* used in the program is denoted by the countable set $\mathbb{Z}$, and v will range over values.

Definition 1.1.5. The set of *environments* is denoted $\mathbf{Env}_c = \mathbf{Var} \to \mathbb{Z}$, and env_c will range over these "concrete" environments.

Although numeric representations within computers are typically bounded (in my computer an integer can range between -2^{31} and $2^{32}-1$), there are other data types which are not bounded, e.g. lists. Some languages even have access to arbitrary precision numbers for which there are no bounds (disregarding the physical memory limit of the machine). For this reason we use an infinite set of values.

A program is said to have an initial environment which is used to parameterise the execution. Some of the variables used in the program are mapped to initial values, the rest are undefined and assumed to be assigned to before they are used. (The safety of the language is not relevant for a summary of Abstract Interpretation.)

Definition 1.1.6. The initial environment is denoted $\mathbf{env}_0 \in \mathbf{Env}_c$.

We define an arithmetic and boolean expression syntax. We range over arithmetic and boolean expressions with $e \in \mathbf{AExp}$ and $b \in \mathbf{BExp}$ respectively.

Definition 1.1.7. Syntax of arithmetic and boolean expressions:

$$e ::= x_1 + x_2 \mid x_1 * x_2$$
$$b ::= x \geq 0 \mid x < 0$$

We also have a semantic function for evaluating arithmetic expressions in the context of an environment, and likewise a semantic function for evaluating boolean expressions.

Definition 1.1.8. Semantic functions for evaluation:

$$\mathcal{A}_c : \mathbf{AExp} \to \mathbf{Env}_c \to \mathbb{Z}$$
$$\mathcal{B}_c : \mathbf{BExp} \to \mathbf{Env}_c \to \{\mathbf{True}, \mathbf{False}\}$$

Arithmetic evaluation is only done in the assignment node, and boolean evaluation only at the test nodes. Note that this is a rather limited syntax for expressions, but it is possible to recover the power of a more complex syntax by using many assignments or tests. Numeric literals can be encoded by defining variables to evaluate to them, these variables never being assigned to in the program. It is possible for create an expression that simply evaluates a variable by using the expression $x + 0$. Our examples may assume extensions to this syntax, for the sake of clarity.

Our semantics requires a notion of environment update, which is formalised as follows:

Definition 1.1.9. Environment *update* is denoted $\mathbf{env}_c[x \mapsto v]$. The precise nature of the environment $\mathbf{env}'_c = \mathbf{env}_c[x \mapsto v]$ is:

$$\mathbf{env}'_c(x') = \begin{cases} \mathbf{env}_c(x) & \text{if } x \neq x' \\ v & \text{otherwise} \end{cases}$$

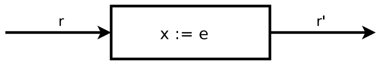
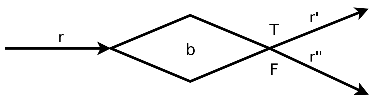
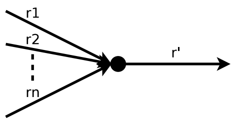
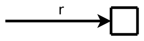
Evaluation Function	Program Structure Condition
$\langle r, \mathbf{env}_c \rangle \rightsquigarrow \langle r', \mathbf{env}_c[x \mapsto \mathcal{A}_c[\![e]\!](\mathbf{env}_c)] \rangle$	
$\langle r, \mathbf{env}_c \rangle \rightsquigarrow \begin{cases} \langle r', \mathbf{env}_c \rangle & \text{if } \mathcal{B}_c[\![b]\!](\mathbf{env}_c) = \textbf{True} \\ \langle r'', \mathbf{env}_c \rangle & \text{if } \mathcal{B}_c[\![b]\!](\mathbf{env}_c) = \textbf{False} \end{cases}$	
$\forall 1 \leq i \leq n \ . \ \langle r_i, \mathbf{env}_c \rangle \rightsquigarrow \langle r', \mathbf{env}_c \rangle$	
$\langle r, \mathbf{env}_c \rangle \rightsquigarrow \mathbf{env}_c$	

Table 1.1: Operational semantics of the flow chart language used in this report.

The execution of the program is formalised with an evaluation function that defines the effect of a single step of execution. Since this is a function and not a general relation, the language is deterministic. If the function returns another pair, the execution is incomplete, otherwise we say the execution has terminated.

Definition 1.1.10. The *evaluation function* $(\rightsquigarrow) : \mathbf{Arc} \times \mathbf{Env}_c \to (\mathbf{Arc} \times \mathbf{Env}_c) \cup \mathbf{Env}_c$.

The complete execution is therefore the transitive closure of this function, starting from $\langle r_0, \mathbf{env}_0 \rangle$ where r_0 is the single entry arc of the program. This can be represented by either a finite sequence of $\mathbf{Arc} \times \mathbf{Env}_c$ pairs followed by a lone environment, or an infinite sequence of $\mathbf{Arc} \times \mathbf{Env}_c$ pairs. If the execution terminates, the *result* of the execution is represented by the final lone environment.

The evaluation function itself is derived from the graph structure of the program, obeying the rules in table 1.1. Note that it could have been possible to consider a single, more general kind of node, one that takes multiple inputs, has multiple outputs, and can change the state. This would have made the semantics rather more complex (although more concise) since selection of an outward arc, updating of state, and merging of input arcs would need to be done in a single expression.

Example 1.1.11. *Factorial:* An example implementation of the factorial algorithm is shown in figure 1.2. If the program is run with parameter $\texttt{n} = 6$ until it stops, the result will be $\texttt{c} = 720$. In terms of the notation, with initial environment satisfying $\mathbf{env}_0(n) = 6$, transitive closure of $\langle r_1, \mathbf{env}_0 \rangle$ will contain an environment $\mathbf{env}_c$ satisfying $\mathbf{env}_c(n) = 1$ and $\mathbf{env}_c(c) = 720$.

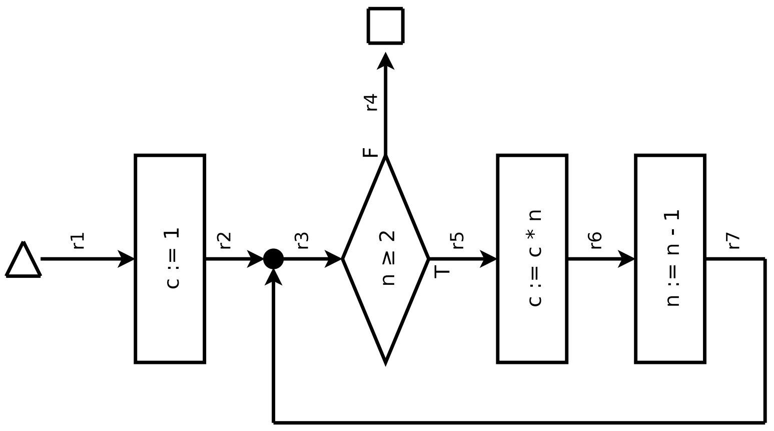


Figure 1.2: The factorial algorithm represented with a flow chart.

1.2 The accumulating semantics as a basis for analysis

The conventional method for defining the semantics of programs (as in the previous section) is concerned with knowing the result (or possibly the effect) of a particular execution of a program. This is not suitable for program analysis for three reasons:

- We need to be assured of properties for *all possible executions*.
- We need information that relates to the various "parts" of the program code, not just at its exit arcs.
- We need not just information relating to the incidental state of a partial execution, but also the past and future of that execution. This is necessary for certain analyses such as available expressions, well-definedness of variables, and liveness.

To address these issues an "accumulating" semantics is defined, that relates to the operational semantics, but retains more information than the progressive mutation of a single environment along a single route of the flow chart. This semantics was originally defined in [4] and termed the "static" semantics. In [7] it is called the "collecting" semantics, whereas in [6] it is called the accumulating semantics, and this is the name used here.

The principle is the same however: We wish to define an analysis that is the basis for all analyses, a middle ground that captures the style of the abstract, imprecise analyses while being sufficiently general as to capture *all* program properties. This section shows how this is possible, and how the accumulating semantics addresses the above issues.

For the first issue, we consider a set of environments, instead of a single environment, at each invocation of the evaluation function. The environments attributed with the entry arc, are therefore the set of all possible initial environments, and we consider the effect of execution on each of them, at each step.

What constitutes a "part" of the program code will vary from language to language. The flow chart representation is attuned for this need – the set of places where we might like to gather information is simply the set of arcs **Arc**.

Often with graphs, we consider nodes to represent stability, and the edges to represent transition. With our flow chart representation of programs however, the opposite is true. The arcs represent a stable environment, and an intermediate location during execution. The nodes represent the transition to the next environment and location. Thus it is natural to collect program properties at each arc. We simply keep a record of what environments existed at each arc during the course of execution.

To record the history of a given execution, we can simply append each new environment on the end of a "trace", and record the traces instead of the environments. During execution we will only refer to the last (the most recent) element of this trace, in order to decide how to proceed, but the history will be there for the purpose of analysis.

The same can be done for the future of the execution, but we have to completely reverse the direction of the semantics, and consider the set of possible exit environments. In this report we deal only with "forwards" analysis, that is gathering information relating to the history of an execution.

In order to combine and implement these ideas formally, we need to define some preliminary notations.

Definition 1.2.1. The set of lists of some set, X, is defined in an inductive manner:

$$\begin{array}{ll} [\ x\ ] \in \mathcal{L}ist(X) & \text{where } x \in X \\ [\ L \mid x\ ] \in \mathcal{L}ist(X) & \text{where } x \in X \text{ and } L \in \mathcal{L}ist(X) \end{array}$$

The top form allows for the construction of a "singleton" list, whereas the bottom notation allows for appending an element to the end of an existing list. The $\overline{L}$ notation allow us to extract the last element from a list, formally:

Definition 1.2.2. Extraction of the last member of a list:

$$\begin{array}{l} \overline{[\ x\ ]} = x \\ \overline{[\ L \mid x\ ]} = x \end{array}$$

We use lists to define "traces" – records of the history of individual executions. A trace records every arc that that the execution flows down, and what environment was present at that time. The last member of the trace represents the current arc together with the current environment – the same information used to determine the next computational step in the operational semantics.

Definition 1.2.3. The set of traces is defined by $\mathbf{Trace} = \mathcal{L}ist(\mathbf{Arc} \times \mathbf{Env}_c)$. The symbol $\mathbf{tr}$ ranges over traces.

Definition 1.2.4. It is helpful to have a concise notation for selecting the environment from a pair of type $\mathbf{Arc} \times \mathbf{Env}_c$. We use ".2", e.g. $\langle r, \mathbf{env}_c \rangle.2 = \mathbf{env}_c$.

It will be common to see the phrase $\overline{\mathbf{tr}}.2$ in this semantics, since this is the most recent environment of a given trace.

We wish to create a semantics which records, for all possible values of input parameters, at all arcs in the program, the set of partial executions that have reached that point. This will require a radically different kind of semantics, but clearly also we will need to consider a set of initial environments rather than the single environment used before, when parameterising the program.

Definition 1.2.5. Initial environments: $\mathbf{envs}_0 \in \mathbb{P}(\mathbf{Env}_c)$

To hold all the information required, the state of the new "computing machine" is ranged over by $c \in C = \mathbf{Arc} \to \mathbb{P}(\mathbf{Trace})$. There is no concept of "current arc"; we consider all arcs and inputs simultaneously. Indeed different paths may be taken by different traces, at a given computational step, depending on the data in each initial environment.

Instead of the transitive closure of a state transition function (as in the operational semantics), we are interested in computing the least fixed point of the equation $c = \mathbf{acc}(c)$. This computing machine has initial state $\lambda r.\emptyset \in C$ and a state transition function of the form $\mathbf{acc} : C \to C$, and as before, the definition of $\mathbf{acc}$ is defined by the program, but this time obeys the rules in table 1.3.

The definition of $\mathbf{acc}$ has a different flavour to the function $(\leadsto)$. Firstly we are obviously dealing with sets of traces instead of individual environments, so we have to maintain and look up information from these traces. Secondly, we translate information relating to all the arcs to *more* information relating to all the arcs. This new information is the traces generated as computations enter the arc by traversing the source node. We are no longer concerned with where a given arc is pointing, when we iterate this machine. Instead we need to know where it has come from. We now consider entry arcs (which we did not before), and we no longer need to consider exit arcs, since there is nothing special about the source of an exit arc.

We can formally describe the relationship between the operational and accumulating semantics with the following statement, it can be seen to hold with the definitions given:

Lemma 1.2.6. *For all c and $\mathbf{env}_c \in \mathbf{envs}_0$, and if the entry arc is r_0:*

$$[\langle r_0, \mathbf{env}_c \rangle] \in \mathbf{acc}(c)(r_0)$$

*Proof: By definition of **acc**.*

Intuitively this means that every possible initial environment will be represented by the start of a trace at the entry arc.

Lemma 1.2.7. *For all $\langle r, \mathbf{env}_c \rangle, \langle r', \mathbf{env}'_c \rangle, c$*

$\langle r, \mathbf{env}_c \rangle \leadsto \langle r', \mathbf{env}'_c \rangle \iff$ both of the following hold:

- $\mathbf{tr} = [\langle r, \mathbf{env}_c \rangle] \in c(r) \implies [\ \mathbf{tr} \mid \langle r', \mathbf{env}'_c \rangle\ ] \in \mathbf{acc}(c)(r)$
- $\mathbf{tr} = [\ \mathbf{tr}' \mid \langle r, \mathbf{env}_c \rangle\ ] \in c(r) \implies [\ \mathbf{tr} \mid \langle r', \mathbf{env}'_c \rangle\ ] \in \mathbf{acc}(c)(r)$

*Proof: By case analysis of $(\leadsto)$ and definition of **acc**.*

$$
\mathbf{acc}(c) = \lambda r.\begin{cases}
\{\ \ [\ \langle r, \mathbf{env}_c\rangle\ ]\ \mid\ \mathbf{env}_c \in \mathbf{envs}_0\ \ \} & \text{if } (1)\\
\{\ \ [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2[x \mapsto \mathcal{A}_c[\![e]\!]\overline{\mathbf{tr}}.2]\rangle\ ]\ \mid\ \mathbf{tr} \in c(r_p)\ \ \} & \text{if } (2)\\
\{\ \ [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ]\ \mid\ \mathbf{tr} \in c(r_p)\ \wedge\ \mathcal{B}_c[\![b]\!]\overline{\mathbf{tr}}.2 = \mathbf{True}\ \} & \text{if } (3)\\
\{\ \ [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ]\ \mid\ \mathbf{tr} \in c(r_p)\ \wedge\ \mathcal{B}_c[\![b]\!]\overline{\mathbf{tr}}.2 = \mathbf{False}\ \} & \text{if } (4)\\
\bigcup_{i=1}^{n}\{\ \ [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ]\ \mid\ \mathbf{tr} \in c(r_i)\ \ \} & \text{if } (5)
\end{cases}
$$

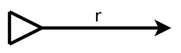
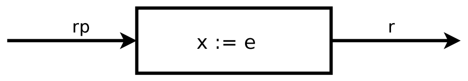
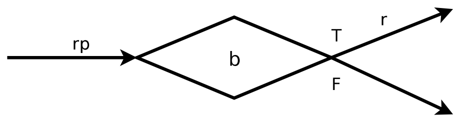
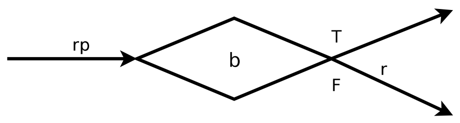
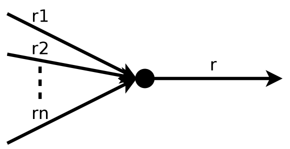
(1)	
(2)	
(3)	
(4)	
(5)	

Table 1.3: Definition of the state transfer function for the accumulating semantics.

Intuitively this means that every computational step from traces accumulated in the computing machine's state will be accounted for when **acc** is next called. The reason for the two bulleted expressions is that there are two possibilities for the formation of a trace within $c(r)$, it is a minor notational inconvenience and should not distract from the point being made:

If $\mathbf{acc}(c) = c$, all possible execution steps (and thus the entire execution for each possible initial environment) have been accounted for, and the collection of execution information is complete. This is why we are interested in the least fixed point. It is important to realise that a property represents a possibility during execution. It is safe, although not accurate, to suggest that more properties might apply, than actually do. The converse on the other hand, is completely unacceptable. A fixed point represents safety, since all the necessary properties have been accounted for. The least fixed point is the most accurate.

Theorem 1.2.8. *All and the only possible partial computation sequences in the operational semantics for a program, starting with an initial environment from* $\boldsymbol{envs}_0$*, are each represented as a trace at the appropriate arc in the least fixed point of* **acc** *for that program. Proof: induction on the length of computation sequences, using lemmas 1.2.6 and 1.2.7.*

Example 1.2.9. *Factorial:* Recall the implementation of the factorial algorithm shown in figure 1.2. With initial environments assigning the variable `n` to be 2, 3 and 4, the least fixed point of **acc** is shown in figure 1.4. The format of the environment in the traces has been invented for conciseness: If an environment maps `n` to 1 and `c` to 2, the format in the figure would be `<01,02>`. `Ud` stands for "undefined".

Recall that the purpose of defining an accumulating semantics was to allow the collection of information that can be analysed to infer any desired program properties. We have shown that the fixed point of the accumulating semantics is capable of providing exactly this information. It reveals the precise detail of all executions that pass through a given arc.

The accumulating semantics has a structure very similar to many analyses, although it is unusual in that it has ultimate precision (and of course may never terminate – there may be no fixed point). The state of the computing machine forms a lattice since **Arc** is finite and the powerset of a set forms a lattice:

```
r1: { [<1,(02,Ud)>],
      [<1,(03,Ud)>],
      [<1,(04,Ud)>] }

r2: { [<1,(02,Ud)> <2,(02,01)>],
      [<1,(03,Ud)> <2,(03,01)>],
      [<1,(04,Ud)> <2,(04,01)>] }

r3: { [<1,(02,Ud)> <2,(02,01)> <3,(02,01)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)>],
      [<1,(02,Ud)> <2,(02,01)> <3,(02,01)> <5,(02,01)> <6,(02,02)> <7,(01,02)> <3,(01,02)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)> <3,(02,03)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)> <3,(02,03)> <5,(02,03)> <6,(02,06)> <7,(01,06)> <3,(01,06)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)> <3,(02,12)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)> <3,(02,12)> <5,(02,12)> <6,(02,24)> <7,(01,24)> <3,(01,24)>] }

r4: { [<1,(02,Ud)> <2,(02,01)> <3,(02,01)> <5,(02,01)> <6,(02,02)> <7,(01,02)> <3,(01,02)> <4,(01,02)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)> <3,(02,03)> <5,(02,03)> <6,(02,06)> <7,(01,06)> <3,(01,06)> <4,(01,06)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)> <3,(02,12)> <5,(02,12)> <6,(02,24)> <7,(01,24)> <3,(01,24)> <4,(01,24)>] }

r5: { [<1,(02,Ud)> <2,(02,01)> <3,(02,01)> <5,(02,01)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)> <3,(02,03)> <5,(02,03)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)> <3,(02,12)> <5,(02,12)>] }

r6: { [<1,(02,Ud)> <2,(02,01)> <3,(02,01)> <5,(02,01)> <6,(02,02)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)> <3,(02,03)> <5,(02,03)> <6,(02,06)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)> <3,(02,12)> <5,(02,12)> <6,(02,24)>] }

r7: { [<1,(02,Ud)> <2,(02,01)> <3,(02,01)> <5,(02,01)> <6,(02,02)> <7,(01,02)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)>],
      [<1,(03,Ud)> <2,(03,01)> <3,(03,01)> <5,(03,01)> <6,(03,03)> <7,(02,03)> <3,(02,03)> <5,(02,03)> <6,(02,06)> <7,(01,06)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)>],
      [<1,(04,Ud)> <2,(04,01)> <3,(04,01)> <5,(04,01)> <6,(04,04)> <7,(03,04)> <3,(03,04)> <5,(03,04)> <6,(03,12)> <7,(02,12)> <3,(02,12)> <5,(02,12)> <6,(02,24)> <7,(01,24)>] }
```

Figure 1.4: Fixed point of the accumulating semantics for the factorial example.

Theorem 1.2.10. *The accumulating semantics is a monotone framework:*

Set:	$C = \mathbf{Arc} \to \mathbb{P}(\mathbf{Trace})$
Partial order:	$c_1 \sqsubseteq_c c_2 \iff \forall r \ . \ c_1(r) \subseteq c_2(r)$
Least upper bound:	if $C' \subseteq C$, $\sqcup_c C' = \lambda r. \bigcup\{c(r) \mid c \in C'\}$
Greatest lower bound:	if $C' \subseteq C$, $\sqcap_c C' = \lambda r. \bigcap\{c(r) \mid c \in C'\}$
Top:	$\top_c = \lambda r.\mathbf{Trace}$
Bottom:	$\bot_c = \lambda r.\emptyset$
Monotone transfer function:	**acc**

Proof: Lemma 1.2.11 and the rest is straightforward because the lattice C is a pointwise lifting of a powerset lattice.

Lemma 1.2.11. *Monotonicity of* ***acc****:* $c_1 \sqsubseteq c_2 \Longrightarrow \mathbf{acc}(c_1) \sqsubseteq \mathbf{acc}(c_2)$

Proof: sufficient to prove for expanded $(\sqsubseteq)$. *Note case (1) of* $\mathbf{acc}(c)(r)$ *is trivial since it is constant with respect to c.*

$$
\begin{array}{|ll}
\text{Prove } \forall r, \mathbf{tr} \in \mathbf{acc}(c_1)(r).\mathbf{tr} \in \mathbf{acc}(c_2)(r). & \\
\hline
\text{For all } r, \text{ let } \mathbf{tr} \in \mathbf{acc}(c_1)(r): & \\
\mathbf{tr} \in \mathbf{acc}(c_1)(r) & \Rightarrow \\
\text{case (2-4):} & \\
\mathbf{tr} \in \{ \ \dots \mid \mathbf{tr'} \in c_1(r_p) \ \wedge \ \dots \} & \Rightarrow \\
\mathbf{tr} \in \{ \ \dots \mid \mathbf{tr'} \in c_2(r_p) \ \wedge \ \dots \} & \Rightarrow \\
\mathbf{tr} \in \mathbf{acc}(c_2)(r) & \square \\
\text{case (5):} & \\
\mathbf{tr} \in \bigcup_{i=1}^{n} \{ \ \dots \mid \mathbf{tr'} \in c_1(r_i) \ \} & \Rightarrow \\
\exists i.\mathbf{tr} \in \{ \ \dots \mid \mathbf{tr'} \in c_1(r_i) \ \} & \Rightarrow \\
\exists i.\mathbf{tr} \in \{ \ \dots \mid \mathbf{tr'} \in c_2(r_i) \ \} & \Rightarrow \\
\mathbf{tr} \in \bigcup_{i=1}^{n} \{ \ \dots \mid \mathbf{tr'} \in c_2(r_i) \ \} & \Rightarrow \\
\mathbf{tr} \in \mathbf{acc}(c_2)(r) & \square
\end{array}
$$

Intuitively, **acc** has to be monotone, since otherwise we would potentially lose trace information as the analysis proceeds, and this would compromise the *safety* of the result (the relationship with the operating semantics).

This concludes the study of the accumulating semantics. We use this theory heavily in chapter 2, as a foundation from which analyses can be derived.

1.3 An example analysis

One can think of the accumulating semantics as a "very precise" analysis, but we are interested in imprecise analyses, with useful termination properties. This chapter gives an example of an imprecise analysis, although there is no proof yet that it is without error.

In chapter 2 we will derive this very analysis from the accumulating semantics using mathematical notation. First, however, it helps to simply state the analysis, as it may be designed by an engineer, so we can see what it is we hope to reach.

The chosen analysis is a sign analysis. The technique is to abstract the computation to only consider whether a variable is positive ≥ 0 or negative < 0, or potentially may be either, at each arc. We also need to represent the possibility that we know nothing about a variable, that is we have seen no evidence that it is positive or negative. Should an analysis terminate in this state for a variable on an arc, it will mean that either the variable was never defined, or the arc is never traversed.

Definition 1.3.1. The possible *abstract values* for a variable: $\mathbf{Sign} = \mathbb{P}(\{+,-\}) = \{\emptyset, +, -, \pm\}$. This set is ordered with the subset relation and is therefore a lattice.

Definition 1.3.2. We use the function s to determine the sign of a value $v \in \mathbb{Z}$:

$$s(v) = \begin{cases} + & \text{if } v \geq 0 \\ - & \text{if } v < 0 \end{cases}$$

Definition 1.3.3. The *abstract environment*: $\mathbf{Env}_a = \mathbf{Var} \to \mathbf{Sign}$, $\mathbf{env}_a$ ranges over $\mathbf{Env}_a$.

Definition 1.3.4. The *abstract state* of the analysis's computing machine: $A = \mathbf{Arc} \to \mathbf{Env}_c$, the symbol a ranges over these states as c ranged over C in the accumulating semantics.

We will need a few more preliminaries for the definition of the state transfer function, **int** as defined in 1.5. We use the *same* set of initial environments as for the accumulating semantics, since this information is part of the program itself rather than part of the semantics of the analysis. The conversion from concrete environments to abstract environments is handled by **int** itself. The analysis proceeds by finding the least fixed point of the function **int** just as with the accumulating semantics. The initial state of the analysis is $\lambda r \lambda x.\emptyset$.

Definition 1.3.5. *Evaluation of arithmetic and boolean expressions* in abstract environments: $\mathcal{A}_a : \mathbf{AExp} \to \mathbf{Env}_a \to \mathbf{Sign}$ and $\mathcal{B}_a : \mathbf{BExp} \to \mathbf{Env}_a \to \mathbb{P}(\{\mathbf{True}, \mathbf{False}\})$ is defined by the following exhaustive tables:

$\mathcal{A}_a[\![x_1 + x_2]\!]$	$\emptyset$	$+$	$-$	$\pm$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$+$	$\emptyset$	$+$	$\pm$	$\pm$
$-$	$\emptyset$	$\pm$	$-$	$\pm$
$\pm$	$\emptyset$	$\pm$	$\pm$	$\pm$

$\mathcal{A}_a[\![x_1 * x_2]\!]$	$\emptyset$	$+$	$-$	$\pm$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$+$	$\emptyset$	$+$	$-$	$\pm$
$-$	$\emptyset$	$-$	$+$	$\pm$
$\pm$	$\emptyset$	$\pm$	$\pm$	$\pm$

$\mathcal{B}_a[\![b]\!]$	$b \equiv x \geq 0$	$b \equiv x < 0$
$\emptyset$	$\emptyset$	$\emptyset$
$+$	$\{\mathbf{True}\}$	$\{\mathbf{False}\}$
$-$	$\{\mathbf{False}\}$	$\{\mathbf{True}\}$
$\pm$	$\{\mathbf{True}, \mathbf{False}\}$	$\{\mathbf{True}, \mathbf{False}\}$

$$\mathbf{int}(a) = \lambda r. \begin{cases} \lambda x.\{ \;\; s(env_c)(x) \;\; | \;\; \mathbf{env}_c \in \mathbf{envs}_0 \;\; \} & \text{if } (1) \\ a(r_p)[\; x \mapsto \mathcal{A}_a[\![e]\!](a(r_p)) \;] & \text{if } (2) \\ \lambda x. \bigcup \{ \; \mathbf{env}_a(x) \mid (\forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')) \;\wedge\; \mathbf{True} \in \mathcal{B}_a[\![b]\!]\mathbf{env}_a \; \} & \text{if } (3) \\ \lambda x. \bigcup \{ \; \mathbf{env}_a(x) \mid (\forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')) \;\wedge\; \mathbf{False} \in \mathcal{B}_a[\![b]\!]\mathbf{env}_a \; \} & \text{if } (4) \\ \lambda x. \bigcup_{i=1}^{n} a(r_i)(x) & \text{if } (5) \end{cases}$$

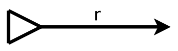
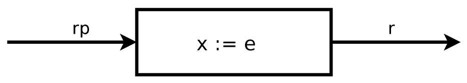
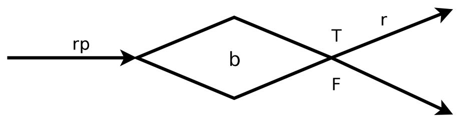
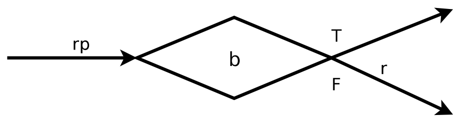
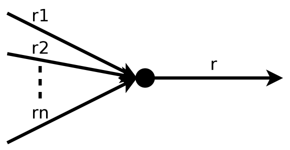
(1)	
(2)	
(3)	
(4)	
(5)	

Table 1.5: Definition of the state transfer function for the sign analysis.

The analysis is a monotone framework just like the accumulating semantics, but since the lattice is finite, every ascending chain will also be finite. Thus the computation of the least fixed point by iteration of **int**, starting from $\bot_a$, will always terminate in finite time.

Definition 1.3.6. *Sign analysis, as a monotone framework:*

Set:	$A = \mathbf{Arc} \to \mathbf{Env}_a$
Complete partial order:	$a_1 \sqsubseteq_a a_2 \iff \forall r, x \ . \ a_1(r)(x) \subseteq a_2(r)(x)$
Least upper bound:	if $A' \subseteq A$, $\sqcup_a A' = \lambda r \lambda x. \bigcup\{a(r)(x) \vert a \in A'\}$
Greatest lower bound:	if $A' \subseteq A$, $\sqcap_a A' = \lambda r \lambda x. \bigcap\{a(r)(x) \vert a \in A'\}$
Top:	$\top_a = \lambda r.\lambda x.\pm$
Bottom:	$\bot_a = \lambda r \lambda x.\emptyset$
Monotone transfer function:	**int**

Proof: The monotonicity of ***int****, proceeds in a very similar manner to lemma 1.2.11. The rest is straight forward since the lattice A is a powerset lattice lifted to two finite sets.*

Example 1.3.7. *An example of sign analysis:* Sign analysis is not interesting when applied to the factorial example, so instead we consider a new example program, shown in figure 1.6. We consider finite sets of values for the parameters of the program (x and y). The variable x ranges between -10 and -5, whereas y varies between -10 and 10. We iterate the analysis starting from the empty set of properties: $\lambda r \lambda x.\emptyset$.

The progress of the analysis is shown in table 1.7. The format of the abstract environment has been invented for conciseness: If x is positive and y is negative, the displayed abstract environment would be $(+, -)$. Note that the analysis terminates in finite time, whereas the accumulating semantics does not.

It is possible to generalise this analysis to use any kind of finite, “abstract” environment, we have simply chosen to divide the range of values into two sets at zero. Any finite set of equivalence classes on $\mathbb{Z}$ would do for ensuring termination, e.g. multiples of a specific number (parity analysis).

In addition, we are not looking at any history or future information in the analysis, such as might be done with a liveness analysis. The sign analysis was chosen because it is simple and has an interesting behaviour with test nodes for our simple language.

In the next chapter we bring the accumulating semantics and the proposed analysis together, show that the analysis can be derived from a simple specification, that it is safe with respect to the accumulating semantics, and also that it is the most accurate sign analysis possible.

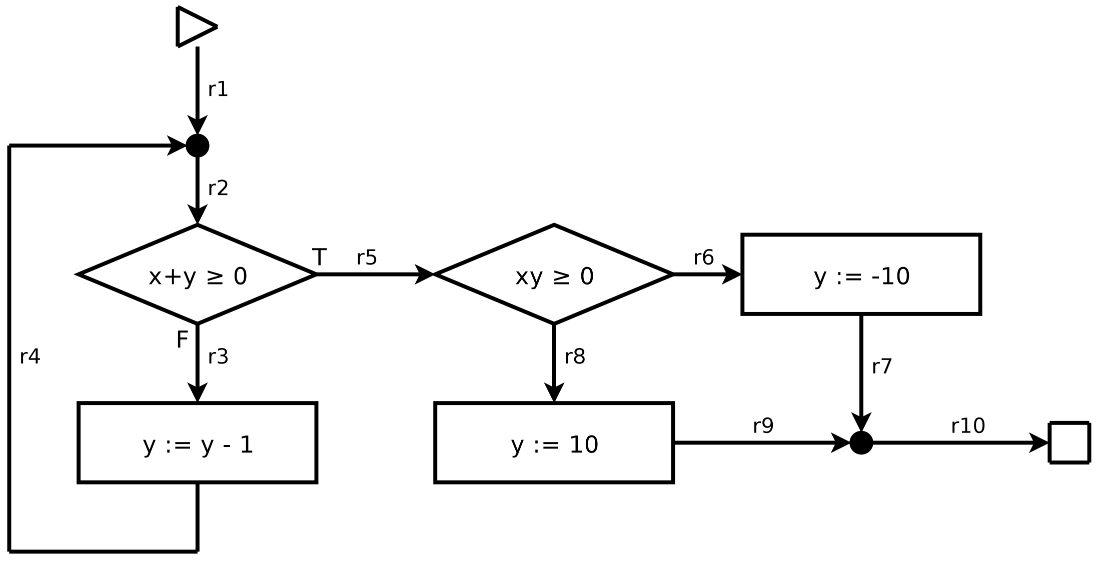


Figure 1.6: An example program for the sign analysis.

Arc	0	1	2	3	4	5	6 (FP)
r_1	$(\emptyset,\emptyset)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$
r_2	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$
r_3	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$
r_4	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,\pm)$	$(-,\pm)$	$(-,\pm)$
r_5	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,+)$	$(-,+)$	$(-,+)$	$(-,+)$
r_6	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$
r_7	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$
r_8	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,+)$	$(-,+)$	$(-,+)$
r_9	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,+)$	$(-,+)$
r_{10}	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(-,+)$

Table 1.7: The results of the sign analysis, shown step by step up until the fixed point.

Chapter 2

Abstract Interpretation

In this chapter it is shown how one can derive the sign analysis from the accumulating semantics with a simple specification. Recall that we have two lattices, A and C, representing the abstract and concrete worlds respectively. We have not yet defined a relationship between them, and we have no formal description of what the essence of the sign analysis is, from which to build the rest of the analysis.

2.1 The link between the accumulating semantics and an imprecise analysis

Both of these problems can be solved by specifying a pair of functions, α and γ, that translate meanings from each world to the other.

Definition 2.1.1. *Abstraction and concretisation functions:*

$$\begin{aligned} \alpha &: C \to A \\ \gamma &: A \to C \end{aligned}$$

The definition of γ is rather complex, since it has to pick up the pieces after information has been necessarily lost, but α should be intuitive to understand. Later we will show how to specify γ in terms of α. Recall that $\overline{\mathbf{tr}}.2$ selects the most recent environment from a trace.

Definition 2.1.2. *The sign analysis abstraction and concretisation functions:*

$$\begin{aligned} \alpha(c) &= \lambda r \lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in c(r)\ \} \\ \gamma(a) &= \lambda r.\{\ \mathbf{tr} \mid \forall x\ .\ s(\overline{\mathbf{tr}}.2(x)) \in a(r)(x)\ \} \end{aligned}$$

It is now evident that we can define a transfer function for our sign analysis lattice that represents the imprecise analysis, using the accumulating semantics, and α and γ. All that is required to build the analysis, is the specification of the program properties we are interested in (the lattice), and their meaning, in terms of the accumulating semantics (α and γ). We will compute the least fixed point of this function, using an iterative algorithm starting from $\perp_a$.

Definition 2.1.3. *Sign analysis using abstract interpretation:*

Set:	$A = \mathbf{Arc} \rightarrow \mathbf{Env}_a$
Complete partial order:	$a_1 \sqsubseteq_a a_2 \iff \forall r, x \ . \ a_1(r)(x) \subseteq a_2(r)(x)$
Least upper bound:	if $A' \subseteq A$, $\sqcup_a A' = \lambda r \lambda x. \bigcup\{a(r)(x) \vert a \in A'\}$
Greatest lower bound:	if $A' \subseteq A$, $\sqcap_a A' = \lambda r \lambda x. \bigcap\{a(r)(x) \vert a \in A'\}$
Top:	$\top_a = \lambda r. \lambda x. \pm$
Bottom:	$\bot_a = \lambda r \lambda x. \emptyset$
Monotone transfer function:	$\lambda a. \alpha(\mathbf{acc}(\gamma(a)))$

We must still ensure that the bounds operators, and the top and bottom are indeed what they claim to be. For this abstract interpretation to work correctly, α and γ cannot simply be any function, they must satisfy some properties to make this work. We expect the two functions to be complementary, so we want $\alpha \circ \gamma$ to be the identity function, but if $\gamma \circ \alpha$ is the identity function, then no precision can be lost by representation in the abstract world. We need this imprecision since it allows us to use a finite A, and get guaranteed termination of the analysis.

We do, however, want to make sure that $\gamma \circ \alpha$ will never result in a less "safe" accumulating state, where safety here is the the relation $\sqsubseteq$. We need to be able to safely move between the abstract and concrete worlds.

We also need α and γ to be monotone, so that the information in A is ordered in a consistent way to C, that is that an approximation of safer data (with respect to $\sqsubseteq_c$), is considered safer with respect to $\sqsubseteq_a$. This also gives us the monotonicity of $\lambda a. \alpha(\mathbf{acc}(\gamma(a)))$ which we need for termination within the monotone framework, or intuitively – through the course of abstract interpretation we do not lose information that we have discovered in previous iterations.

Definition 2.1.4. *Monotonicity requirements for α and γ:*

α is monotone:	$\forall c_1 \sqsubseteq_c c_2 \ . \ \alpha(c_1) \sqsubseteq_a \alpha(c_2)$
γ is monotone:	$\forall a_1 \sqsubseteq_a a_2 \ . \ \gamma(a_1) \sqsubseteq_c \gamma(a_2)$

Definition 2.1.5. *Safety for α and γ:*

No precision lost through concretisation:	$\forall a. \alpha(\gamma(a)) = a$
Precision lost safely through abstraction:	$\forall c. \gamma(\alpha(c)) \sqsupseteq_c c$

Theorem 2.1.6. *Sign analysis α and γ are appropriate:*

Proof: Monotonicity for α and γ can be seen immediately. Intuitively, *more traces means more signs* and *vice versa* for α and γ respectively. Rest below:

$$
\begin{array}{|lr}
\text{Prove: } \forall a.\alpha(\gamma(a)) = a & \\
\hline
\lambda r\lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in (\lambda r'.\{\mathbf{tr}' \mid \forall x'.s(\overline{\mathbf{tr}'}.2(x')) \in a(r')(x')\})(r)\ \} & = \\
\lambda r\lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in \{\mathbf{tr}' \mid \forall x'.s(\overline{\mathbf{tr}'}.2(x')) \in a(r)(x')\}\ \} & = \\
\lambda r\lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid \forall x'.s(\overline{\mathbf{tr}}.2(x')) \in a(r)(x')\ \} & = \\
\lambda r\lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid s(\overline{\mathbf{tr}}.2(x)) \in a(r)(x)\ \} & = \\
\lambda r\lambda x.\{\ u \mid u \in a(r)(x)\ \} & = \\
\lambda r\lambda x.a(r)(x) & = \\
a & \square
\end{array}
$$

$$
\begin{array}{|lr}
\text{Prove: } \forall c, \mathbf{tr}, r\ .\ \mathbf{tr} \in c(r) \Longrightarrow \mathbf{tr} \in \gamma(\alpha(c))(r) & \\
\hline
\mathbf{tr} \in c(r) & \Rightarrow \\
\forall x.s(\overline{\mathbf{tr}}.2(x)) \in \alpha(c)(r)(x) & \Rightarrow \\
\mathbf{tr} \in \gamma(\alpha(c))(r) & \square
\end{array}
$$

There is an area of mathematics that considers precisely this state of affairs. If α and γ satisfy the above properties, then we have a Galois insertion, which is a specific kind of Galois connection. In the next section we summarise some useful properties of Galois insertions and connections.

2.2 Galois insertions and Galois connections

2.2.1 Galois connections

Usually a Galois connection (sometimes called an adjunction) is specified in the more concise form:

Definition 2.2.1. *Galois connection, for all a and c:*

$$a \sqsubseteq_a \alpha(c) \Longleftrightarrow \gamma(a) \sqsubseteq_c c$$

We can prove that this definition holds for the requirements in 2.1.4 and 2.1.5: (This proof from [7].)

Theorem 2.2.2. *α and γ are a Galois connection:*

$$
\begin{array}{|lr}
\text{Prove: } a \sqsubseteq_a \alpha(c) \Longrightarrow \gamma(a) \sqsubseteq_c c & \\
\hline
a \sqsubseteq_a \alpha(c) & (2.1.4) \Rightarrow \\
\gamma(a) \sqsubseteq_c \gamma(\alpha(c)) & (2.1.5) \Rightarrow \\
\gamma(a) \sqsubseteq_c c & \square
\end{array}
\quad
\begin{array}{|lr}
\text{Prove: } a \sqsubseteq_a \alpha(c) \Longleftarrow \gamma(a) \sqsubseteq_c c & \\
\hline
\gamma(a) \sqsubseteq_c c & (2.1.4) \Rightarrow \\
\alpha(\gamma(a)) \sqsubseteq_a \alpha(c) & (2.1.5) \Rightarrow \\
a \sqsubseteq_a \alpha(c) & \square
\end{array}
$$

But the best we can prove in the other direction (assuming α and γ are a Galois connection and trying to derive 2.1.4 and 2.1.5) is that $\forall a.\alpha(\gamma(a)) \sqsubseteq_a a$, or intuitively, that C is an approximation of A! (We still have the other properties.) The proof for this is also in [7], but is similar to the above, so is omitted here. This relaxation of the properties of $\alpha \circ \gamma$) is precisely the difference between the Galois connection and the Galois insertion, which the next sub-section explores.

2.2.2 Galois insertion

In a Galois insertion, A approximates C, but in a Galois connection, they are both approximations of each other. For the purposes of abstract interpretation, we have no use for C approximating A, so we want to build a Galois insertion.

It is possible to convert a Galois connection between A and C to a Galois insertion simply by removing the elements from A that are not represented by elements in C. This is illustrated by the following fact:

Theorem 2.2.3. *When α and γ are a Galois connection, they are a Galois insertion if and only if α is surjective:*

Prove: $\alpha(\gamma(a)) = a \Longrightarrow \exists c.\alpha(c) = a$
Simply take $c = \gamma(a)$, which must exist since γ is a total function. $\square$

Prove: $\alpha(\gamma(a)) = a \Longleftarrow \exists c.\alpha(c) = a$	
$\alpha(c) = a$	(mono γ) $\Rightarrow$
$\gamma(\alpha(c)) = \gamma(a)$	(GC) $\Rightarrow$
$\gamma(a) \sqsubseteq_c c$	(mono α) $\Rightarrow$
$\alpha(\gamma(a)) \sqsubseteq_a \alpha(c)$	$\Rightarrow$
$\alpha(\gamma(a)) \sqsubseteq_a a$	(GC) $\Rightarrow$
$\alpha(\gamma(a)) = a$	$\square$

We need only re-define the lattice A so that the only elements remaining are those in the range of α, and we have a Galois insertion. To see this more formally, we must define a $\hat{A}$, $\hat{\sqsubseteq}_a$, and Galois insertion $\hat{\alpha}$ $\hat{\gamma}$ based on Galois connection γ α.

Definition 2.2.4. *Refined state $\hat{A}$ and associated Galois insertion:* (The symbol $\hat{a}$ ranges over $\hat{A}$.)

$$\begin{array}{l}
\hat{A} = \{\ a \mid \alpha(c) = a\ \} \\
\hat{a}_1 \hat{\sqsubseteq}_a \hat{a}_2 \Longleftrightarrow \hat{a}_1 \sqsubseteq_a \hat{a}_2 \\
\forall c\ .\ \hat{\alpha}(c) = \alpha(c) \text{ (i.e. } \alpha \text{ and } \hat{\alpha} \text{ are identical)} \\
\forall \hat{a}\ .\ \hat{\gamma}(\hat{a}) = \gamma(\hat{a})
\end{array}$$

From these definitions we can show that $\hat{\alpha}$ $\hat{\gamma}$ is a Galois connection, and also since $\hat{\alpha}$ is surjective, a Galois insertion.

To see that the set $\hat{A}$ is still a lattice, the monotonicity of $\hat{\alpha}$ and the fact that C is still a lattice means that there will be an upper and lower bound of any subset of $\hat{A}$ at $\hat{\alpha}(\top_c)$ and $\hat{\alpha}(\bot_c)$ respectively. In practice, it will be common to define an α that is surjective, but this shows that a conversion can be made in all cases where this is not so.

2.2.3 Systematic construction of Galois connections

In [7] is described a number of ways in which Galois connections can be formed by connecting together other Galois connections. This means that results from simple "building block" abstract interpretations can be composed without having to prove that the result is valid. This could, in principle, be used by a compiler to autonomously assemble an appropriate abstract interpretation that is guaranteed to be sound and complete.

2.2.4 Defining γ from α

A very useful property of a Galois connection is that γ can be derived from α. We only need to invent the more intuitive α, since we can simply derive γ from this. This also shows that there is only one adjoining γ for α.

Theorem 2.2.5. *If α and γ are a Galois connection, then:* $\gamma(c) = \bigsqcup_c \{c | \alpha(c) \sqsubseteq_a a\}$

> Proof: Clearly, $\gamma(c) = \bigsqcup_c \{c | c \sqsubseteq_c \gamma(c)\}$, but then since α and γ are a Galois connection and $c \sqsubseteq_c \gamma(c) \Leftrightarrow \alpha(c) \sqsubseteq_a a$, it follows that $\gamma(c) = \bigsqcup_c \{c | \alpha(c) \sqsubseteq_a a\}$.

Unfortunately it is not the case that any function α can be found an appropriate γ to form a Galois connection, there are some α that are unsuitable. Ideally, we want a method whereby it is possible to ensure α is definitely *half* of a Galois connection, without reference to any γ. This is where the following result comes in useful. It means if α is completely additive then it is part of a Galois connection, and thus γ can be found as above. The following proof is from [7]:

Theorem 2.2.6. *If and only if α and γ are a Galois connection, then α is completely additive. For all $A' \subseteq A, c$:*

$$
\begin{array}{|lr}
\text{Prove: } \alpha(\bigsqcup_a(A')) \sqsubseteq_c c \Longleftrightarrow \bigsqcup_c\{\alpha(a)|a \in A'\} \sqsubseteq_c c & \\
\hline
\alpha(\bigsqcup_a(A')) \sqsubseteq_c c & (\text{GC}) \Leftrightarrow \\
\bigsqcup_a(A') \sqsubseteq_a \gamma(c) & \Leftrightarrow \\
\forall a \in A'.a \sqsubseteq_a \gamma(c) & (\text{GC}) \Leftrightarrow \\
\forall a \in A'.\alpha(a) \sqsubseteq_c c & \Leftrightarrow \\
\bigsqcup_a\{\alpha(a)|a \in A'\} \sqsubseteq_c c & \Box
\end{array}
$$

Now we will use this result to define γ from α in the sign analysis, by first proving α is completely additive. Unfortunately proofs about the sign analysis, despite it being quite simple, are rather long and technical, even this simple result.

Lemma 2.2.7. *α is completely additive.* Recall that $\alpha(c) = \lambda r \lambda x.\{s(\overline{\mathbf{tr}}.2(x))|\mathbf{tr} \in c(r)\}$. For all $C' \subseteq C$:

$$
\begin{array}{|lr}
\text{Prove: } \forall a, x, u \;.\; u \in \alpha(\bigsqcup_c C')(r)(x) \Longleftrightarrow u \in \bigsqcup_a\{\alpha(c')|c' \in C'\}(r)(x) & \\
\hline
u \in \alpha(\bigsqcup_c C')(r)(x) & \Leftrightarrow \\
u \in \bigcup(\{s(\overline{\mathbf{tr}}.2(x))|\mathbf{tr} \in (\bigsqcup_c C')(r)\}) & \Leftrightarrow \\
\exists \mathbf{tr} \in (\bigsqcup_c C')(r) \;.\; u = s(\overline{\mathbf{tr}}.2(x)) & \Leftrightarrow \\
\exists c' \in C', \mathbf{tr} \in c'(r) \;.\; u = s(\overline{\mathbf{tr}}.2(x)) & \Leftrightarrow \\
\exists c' \in C', u \in \{s(\overline{\mathbf{tr}}.2(x))|\mathbf{tr} \in c'(r)\} & \Leftrightarrow \\
\exists c' \in C', u \in \alpha(c')(r)(x) & \Leftrightarrow \\
u \in \bigcup\{\alpha(c')(r)(x)|c' \in C'\} & \Leftrightarrow \\
u \in \bigsqcup_a\{\alpha(c')|c' \in C'\}(r)(x) & \Box
\end{array}
$$

Theorem 2.2.8. *γ, as defined in terms of α, resolves to the definition given above:*

$$\begin{array}{|ll}
\text{Prove: } \bigsqcup_c\{\ c \mid \alpha(c) \sqsubseteq_a a\ \} = \lambda r.\{\ \mathbf{tr} \mid \forall x\ .\ s(\overline{\mathbf{tr}}.2(x)) \in a(r)(x)\ \} & \\
\hline
\bigsqcup_c\{\ c \mid \alpha(c) \sqsubseteq_a a\ \} & = \\
\bigsqcup_c\{\ c \mid (\lambda r'\lambda x.\{s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in c(r')\}) \sqsubseteq_a a\ \} & = \\
\bigsqcup_c\{\ c \mid (\forall r', x\ .\ \{s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in c(r')\} \subseteq a(r')(x)\ \} & = \\
\lambda r.\bigcup\{\ c(r) \mid (\forall r', x\ .\ \{s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in c(r')\} \subseteq a(r')(x)\ \} & = \\
\lambda r.\bigcup\{\ c(r) \mid (\forall x\ .\ \{s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in c(r)\} \subseteq a(r)(x)\ \} & = \\
\lambda r.\bigcup\{\ \mathbf{trs} \mid \forall x\ .\ \{s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in \mathbf{trs}\} \subseteq a(r)(x)\ \} & = \\
\lambda r.\bigcup\{\ \mathbf{trs} \mid \forall x\ .\ s(\overline{\mathbf{tr}}.2(x)) \in a(r)(x)\ \wedge\ \mathbf{tr} \in \mathbf{trs}\ \} & = \\
\lambda r.\{\ \mathbf{tr} \mid \forall x\ .\ s(\overline{\mathbf{tr}}.2(x)) \in a(r)(x)\ \} & \square
\end{array}$$

Now we have the analysis defined by the monotone framework in 2.1.3, we can show that it is indeed the same as the analysis from section 1.3. In the next section, we will reduce the function $\lambda a.\alpha(\mathbf{acc}(\gamma(a)))$ into the implementable form given earlier.

2.3 Implementing the abstract interpretation

Our current definition of the monotone transfer function, $\lambda a.\alpha(\mathbf{acc}(\gamma(a)))$, is problematic to implement, since it contains infinite sets of traces, and is also very complex compared to the definition of $int : A \to A$ given in section 1.3. However, using standard set theory, we can expand the definitions of α, γ and **acc** to produce a large expression which simplifies into the definition from section 1.3.

$$\alpha(\mathbf{acc}(\gamma(a))) \qquad =$$

We shall use a version of γ from theorem 2.2.8.

$$\alpha(\mathbf{acc}(\lambda r'.\bigcup\{\ \mathbf{trs} \mid \forall x'\ .\ \{s(\overline{\mathbf{tr}}.2(x')) \mid \mathbf{tr} \in \mathbf{trs}\} \subseteq a(r')(x')\ \})) \qquad =$$

Now, we have to deal with **acc** (as defined in table 1.3) on a case for case basis, the most simple case is for entry arcs, (1). This does not even use the result of γ.

$$\alpha(\lambda r'.\{\ [\langle r', \mathbf{env}_c\rangle] \mid \mathbf{env}_c \in \mathbf{envs}_0\ \}) \qquad =(1)$$

And through expansion of α, we get:

$$\begin{array}{l}
\lambda r\lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in (\lambda r'.\{\ [\langle r', \mathbf{env}_c\rangle] \mid \mathbf{env}_c \in \mathbf{envs}_0\ \})(r)\ \} = \\
\lambda r\lambda x.\{\ s(\overline{\mathbf{tr}}.2(x)) \mid \mathbf{tr} \in \{\ [\langle r, \mathbf{env}_c\rangle] \mid \mathbf{env}_c \in \mathbf{envs}_0\ \}\ \}
\end{array} \qquad =(1)$$

Since we only want $\overline{\mathbf{tr}}.2(x)$ from the traces, we can simplify to:

$$\lambda r\lambda x.\{\ s(\mathbf{env}_c(x)) \mid \mathbf{env}_c \in \{\ \mathbf{env}'_c \mid \mathbf{env}'_c \in \mathbf{envs}_0\ \}\ \} \qquad =(1)$$

The second nesting of braces is now redundant:

$$\lambda r \lambda x.\{\ s(\mathbf{env}_c(x)) \mid \mathbf{env}_c \in \mathbf{envs}_0\ \} \qquad \square(1)$$

The next case we consider is (2):

$$\begin{array}{ll} \alpha(\lambda r.\{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2[x \mapsto \mathcal{A}_c[\![e]\!]\overline{\mathbf{tr}}.2]\rangle\ ] \\ \quad \mid & \mathbf{tr} \in (\lambda r'.\bigcup\{\mathbf{trs} \mid \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x')) \mid \mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r')(x')\})(r_p)\ \ \}) \end{array} \qquad =(2)$$

Which simplifies immediately to:

$$\begin{array}{ll} \alpha(\lambda r.\{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2[x \mapsto \mathcal{A}_c[\![e]\!]\overline{\mathbf{tr}}.2]\rangle\ ] \\ \quad \mid & \mathbf{tr} \in \bigcup\{\mathbf{trs} \mid \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x')) \mid \mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_p)(x')\}\ \ \}) \end{array} \qquad =(2)$$

$$\begin{array}{ll} \alpha(\lambda r.\{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2[x \mapsto \mathcal{A}_c[\![e]\!]\overline{\mathbf{tr}}.2]\rangle\ ] \\ \quad \mid & \mathbf{tr} \in \mathbf{trs}\ \wedge\ \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x')) \mid \mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_p)(x')\ \ \}) \end{array} \qquad =(2)$$

Now we expand α (taking care not to clash the x that stands for a given variable, with the specific x that is being assigned to):

$$\begin{array}{ll} \lambda r \lambda x''.\{\ s(\overline{\mathbf{tr}''}.2(x'')) \mid \mathbf{tr}'' \in \{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2[x \mapsto \mathcal{A}_c[\![e]\!]\overline{\mathbf{tr}}.2]\rangle\ ] \mid \mathbf{tr} \in \mathbf{trs}\ \wedge \\ & \forall x'.\{s(\overline{\mathbf{tr}'}.2(x')) \mid \mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_p)(x')\ \ \}\} \end{array} \qquad =(2)$$

And since we are only dealing with $\overline{\mathbf{tr}}.2$:

$$\begin{array}{ll} \lambda r \lambda x''.\{\ s(\mathbf{env}_c(x'')) \mid \mathbf{env}_c \in \{ & \mathbf{env}'_c[x \mapsto \mathcal{A}_c[\![e]\!]\mathbf{env}'_c] \mid \mathbf{env}'_c \in \mathbf{envs}_c\ \wedge \\ & \forall x'.\{s(\mathbf{env}''_c(x')) \mid \mathbf{env}''_c \in \mathbf{envs}_c\} \subseteq a(r_p)(x')\ \}\} \end{array} \qquad =(2)$$

We now deal with the two cases $x'' = x$ and $x'' \neq x$. First the latter case, where, as far as x'' is concerned, $\mathbf{env}'_c[x \mapsto \mathcal{A}_c[\![e]\!]\mathbf{env}'_c] = \mathbf{env}'_c$:

$$\begin{array}{ll} \lambda r \lambda x''.\{\ s(\mathbf{env}_c(x'')) \mid & \mathbf{env}_c \in \mathbf{envs}_c\ \ \wedge \\ & \forall x'\ .\ \{s(\mathbf{env}''_c(x')) \mid \mathbf{env}''_c \in \mathbf{envs}_c\} \subseteq a(r_p)(x')\ \ \} \end{array} \qquad =(2a)$$

And since we are only concerned with x'' in $\mathbf{envs}_c$:

$$\begin{array}{ll} \lambda r \lambda x''.\{\ s(\mathbf{env}_c(x'')) \mid & \mathbf{env}_c \in \mathbf{envs}_c\ \ \wedge \\ & \{s(\mathbf{env}''_c(x'')) \mid \mathbf{env}''_c \in \mathbf{envs}_c\} \subseteq a(r_p)(x'')\ \ \} \end{array} \qquad =(2a)$$

And since we are selecting all members from the set of all objects that obey a certain property, we can reduce this to:

$$\lambda r \lambda x''.\{\ s(\mathbf{env}_c(x'')) \mid s(\mathbf{env}_c(x'')) \in a(r_p)(x'')\ \ \} \qquad =(2a)$$

Substituting $s(\mathbf{env}_c(x''))$ for u, gives:

$$\lambda r \lambda x''.\{\ u \mid u \in a(r_p)(x'')\ \} \qquad =(2a)$$

$$\lambda r \lambda x''.a(r_p)(x'') \qquad =(2a)$$

Since this is the case where $x'' \neq x$:

$$\lambda r \lambda x''.a(r_p)[x \mapsto \mathcal{A}_c[\![e]\!](a(r_p))](x'') \qquad =(2a)$$

Which means the same thing as:

$$\lambda r.a(r_p)[x \mapsto \mathcal{A}_c[\![e]\!](a(r_p))] \qquad \square(2a)$$

And now the harder case, $x'' = x$ where $\mathbf{env}'_c[x \mapsto \mathcal{A}_c[\![e]\!]\mathbf{env}'_c](x'') = \mathcal{A}_c[\![e]\!]\mathbf{env}'_c$.

$$\lambda r \lambda x.\{\ s(\mathcal{A}_c[\![e]\!]\mathbf{env}'_c) \ \mid\ \begin{array}{l} \mathbf{env}'_c \in \mathbf{envs}_c \ \wedge \\ \forall x'.\{s(\mathbf{env}''_c(x'))|\mathbf{env}''_c \in \mathbf{envs}_c\} \subseteq a(r_p)(x')\ \} \end{array} \qquad =(2b)$$

We can introduce another object in the expression, $\mathbf{env}_a$ which is simply an alias: $\mathbf{env}_a = \lambda x''.\{s(\mathbf{env}_c(x))|\mathbf{env}_c \in \mathbf{envs}_c\}$:

$$\lambda r \lambda x.\{\ s(\mathcal{A}_c[\![e]\!]\mathbf{env}'_c) \ \mid\ \begin{array}{l} \mathbf{env}_a = \lambda x''.\{s(\mathbf{env}_c(x))|\mathbf{env}_c \in \mathbf{envs}_c\} \ \wedge \\ \mathbf{env}'_c \in \mathbf{envs}_c \ \wedge \\ \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \} \end{array} \qquad =(2b)$$

It was inevitable that at some stage we would have to deal with the relationship between $\mathcal{A}_c$ and $\mathcal{A}_a$, that is concrete arithmetic evaluation and the abstract approximation of the same. It can be seen, through an exhaustive check of the (finite) $\mathcal{A}_a$ function, that the following expression holds. To understand this expression intuitively, imagine that $\mathbf{env}_a$ defines a set of concrete environments $\mathbf{envs}_c$, and that $\mathcal{A}_a$ examines each member of $\mathbf{envs}_c$ and computes the sign of the concrete evaluation for that environment.

To check this expression, we consider only environments (both $\mathbf{env}_a$ and $\mathbf{env}_c$) that resolve the variables x_1 and x_2 used in the arithmetic expression e. This is easily lifted to the general case, since no other variables affect the result of either concrete, or abstract arithmetic evaluation.

We consider the cases where the set of environments is empty, contains only positive, only negative, or both positive and negative values for each x_1 and x_2, and we must do this for each arithmetic operator. For the abstract environment, this is just the enumeration of all possible cases, and for the concrete environment, it is a partition of the space of possible sets of environments into a finite number of classes.

Lemma 2.3.1. *Concrete and abstract arithmetic evaluation:*

$$\mathcal{A}_a[\![e]\!](\mathbf{env}_a) = \{\ s(\mathcal{A}_c[\![e]\!](\mathbf{env}_c)) \mid \mathbf{env}_a = \lambda x.\{s(\mathbf{env}_c(x))|\mathbf{env}'_c \in \mathbf{envs}_c\}\ \wedge\ \mathbf{env}_c \in \mathbf{envs}_c\ \}$$

This result can now be used to simplify our abstract interpretation to use the abstract arithmetic evaluation operator instead of the concrete one. To make this clearer, the expression is reformulated in the following way:

$$\lambda r \lambda x.\Big\{\ u \ \Big|\ u \in \{\ s(\mathcal{A}_c[\![e]\!]\mathbf{env}_c)\ \ |\ \ \begin{array}{l}\mathbf{env}_a = \lambda x''.\{s(\mathbf{env}_c(x))|\mathbf{env}_c \in \mathbf{envs}_c\} \wedge \\ \mathbf{env}_c \in \mathbf{envs}_c\ \} \wedge\end{array} \\ \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \Big\} \qquad =(2b)$$

This allows the clean replacement of the inner set:

$$\lambda r \lambda x.\{\ u \mid u \in \mathcal{A}_a[\![e]\!](\mathbf{env}_a)\ \wedge\ \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \} \qquad =(2b)$$

Which is the same thing as:

$$\lambda r \lambda x.\bigcup\{\mathcal{A}_a[\![e]\!](\mathbf{env}_a) \mid \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \} \qquad =(2b)$$

Now, in a trick similar to that used at the beginning of lemma 2.2.5, since $\mathcal{A}_a$ is monotonic with respect to the ordering on environments defined by $\forall x.\mathbf{env}_a(x) \subseteq \mathbf{env}'_a(x)$, the union defined above is actually equivalent to $\mathcal{A}_a[\![e]\!]$ applied to the least upper bound of the set of abstract environments, i.e. $a(r_p)$.

$$\lambda r \lambda x.\mathcal{A}_a[\![e]\!](a(r_p)) \qquad =(2b)$$

Since this is the case where $x'' = x$:

$$\lambda r \lambda x''.a(r_p)[x \mapsto \mathcal{A}_c[\![e]\!](a(r_p))](x'') \qquad =(2b)$$

Which means the same thing as:

$$\lambda r.a(r_p)[x \mapsto \mathcal{A}_c[\![e]\!](a(r_p))] \qquad \square(2b)$$

Now for case (3) and (4), which are identical except for the **True** and **False** constraints on the result of boolean expression evaluation on the considered environment.

$$\alpha(\lambda r.\{\ [\mathbf{tr}|\langle r, \overline{\mathbf{tr}}.2\rangle]\ |\ \ \begin{array}{l}\mathbf{tr} \in (\lambda r'.\bigcup\{\ \mathbf{trs} \mid \forall x'.\{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r')(x')\ \})(r_p) \\ \wedge\ \mathcal{B}_c[\![b]\!]\overline{\mathbf{tr}}.2 = \mathbf{True}\ \})\end{array} \qquad =(3,4)$$

Which immediately simplifies to:

$$\alpha(\lambda r.\{\ [\mathbf{tr}|\langle r, \overline{\mathbf{tr}}.2\rangle]\ |\ \ \begin{array}{l}\mathbf{tr} \in \bigcup\{\ \mathbf{trs} \mid \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_p)(x')\ \} \wedge \\ \mathcal{B}_c[\![b]\!]\overline{\mathbf{tr}}.2 = \mathbf{True}\ \})\end{array} \qquad =(3,4)$$

$$\alpha(\lambda r.\{\ [\mathbf{tr}|\langle r, \overline{\mathbf{tr}}.2\rangle]\ |\ \ \begin{array}{l}\mathbf{tr} \in \mathbf{trs}\ \wedge\ \forall x'.\{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_p)(x')\ \} \wedge \\ \mathcal{B}_c[\![b]\!]\overline{\mathbf{tr}}.2 = \mathbf{True}\ \})\end{array} \qquad =(3,4)$$

Now, we expand α

$$\lambda r \lambda x.\{\ s(\overline{\mathbf{tr}''}.2(x)) \mid \mathbf{tr}'' \in \{\ \ \begin{array}{l}[\mathbf{tr}|\langle r, \overline{\mathbf{tr}}.2\rangle] \\ |\ \ \mathbf{tr} \in \mathbf{trs}\ \wedge\ \forall x'.\{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_p)(x')\ \} \wedge \\ \mathcal{B}_c[\![b]\!]\overline{\mathbf{tr}}.2 = \mathbf{True}\quad \}\ \}\end{array} \qquad =(3,4)$$

And as with the previous two cases, we are dealing with environments, not traces, so the record of past execution steps can be stripped away:

$$\begin{aligned}\lambda r\lambda x.\{\ s(\mathbf{env}_c(x))\ \mid\ & \forall x'.\{s(\mathbf{env}'_c(x'))|\mathbf{env}'_c \in \mathbf{envs}_c\} \subseteq a(r_p)(x')\ \wedge \\ & \mathbf{env}_c \in \mathbf{envs}_c\ \wedge\ \mathcal{B}_c[\![b]\!]\mathbf{env}_c = \mathbf{True}\ \}\end{aligned} \qquad =(3,4)$$

We define $\mathbf{env}_a$ as before:

$$\begin{aligned}\lambda r\lambda x.\{\ u\ \mid\ & u \in \mathbf{env}_a(x)\ \wedge\ \mathbf{env}_a = \lambda x''.\{s(\mathbf{env}'_c(x''))|\mathbf{env}'_c \in \mathbf{envs}_c\}\ \wedge \\ & \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \wedge \\ & \mathbf{env}_c \in \mathbf{envs}_c\ \wedge\ \mathcal{B}_c[\![b]\!]\mathbf{env}_c = \mathbf{True}\ \}\end{aligned} \qquad =(3,4)$$

Now it is time to deal with translating the concrete form of boolean evaluation across a potentially infinite set of environments, into the more computable abstract evaluation across a single abstract environment. This is slightly easier than the equivalent arithmetic equivalence, but still very similar. Like before, we inspect the definition of $\mathcal{B}_a$ to ensure the following equivalence is met:

Lemma 2.3.2. *Concrete and abstract boolean evaluation:*

$$\mathcal{B}_a[\![b]\!](\mathbf{env}_a) = \{\ \mathcal{B}_c[\![b]\!](\mathbf{env}_c) \mid \mathbf{env}_a = \lambda x''.\{s(\mathbf{env}'_c(x''))|\mathbf{env}'_c \in \mathbf{envs}_c\}\ \wedge\ \mathbf{env}_c \in \mathbf{envs}_c\ \}$$

Like before, we need only consider environments that define the variables required by the boolean expression, but since boolean expressions are simpler than arithmetic ones, only environments mapping a single variable are considered.

We partition the possible sets of concrete environments into four classes: where there are no environments defined at all, where every environment defines x as positive, likewise where every x is negative, or where there are environments for both polarities within the set. We need to consider the results of boolean evaluation for both kinds of boolean expressions.

Using this identity, we can further simplify the abstract interpretation, but first let us change it so that the substitution is clearer.

$$\begin{aligned}&\lambda r\lambda x.\{\ u\ \mid u \in \mathbf{env}_a(x)\ \wedge\ \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \wedge \\ &\mathbf{True} \in \{\ \mathcal{B}_c[\![b]\!](\mathbf{env}_c) \mid \mathbf{env}_a = \lambda x''.\{s(\mathbf{env}'_c(x''))|\mathbf{env}'_c \in \mathbf{envs}_c\}\ \wedge\ \mathbf{env}_c \in \mathbf{envs}_c\ \}\ \} \\ &=(3,4)\end{aligned}$$

Now we substitute $\mathcal{B}_a[\![b]\!](\mathbf{env}_a)$:

$$\lambda r\lambda x.\{\ u \mid u \in \mathbf{env}_a(x)\ \wedge\ \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \wedge\ \mathbf{True} \in \mathcal{B}_a[\![b]\!](\mathbf{env}_a)\ \} \qquad =(3,4)$$

The last step is to notice that we are collecting together all the members of a set that fits a certain predicate, so this is equivalent to the following:

$$\lambda r\lambda x.\bigcup\{\ \mathbf{env}_a(x) \mid \forall x'.\mathbf{env}_a(x') \subseteq a(r_p)(x')\ \wedge\ \mathbf{True} \in \mathcal{B}_a[\![b]\!](\mathbf{env}_a)\ \} \qquad \square(3,4)$$

The one remaining case is the simpler case of the join node.

$$\begin{array}{ll}\alpha(\lambda r.\{ \bigcup_{i=1}^{n}\{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ] \\ \mid & \mathbf{tr} \in \bigcup\{\ \mathbf{trs} \mid \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_i)(x')\ \}\ \}\}) \end{array} \quad =(5)$$

Flattening the union as with the previous cases:

$$\begin{array}{ll}\alpha(\lambda r.\{ \bigcup_{i=1}^{n}\{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ] \\ \mid & \mathbf{tr} \in \mathbf{trs}\ \wedge\ \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_i)(x')\ \}\}) \end{array} \quad =(5)$$

Now we expand α:

$$\begin{array}{ll}\lambda r \lambda x.\{\ s(\overline{\mathbf{tr}''}.2(x)) \mid \mathbf{tr}'' \in \bigcup_{i=1}^{n}\{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ] \\ \mid & \mathbf{tr} \in \mathbf{trs}\ \wedge \\ & \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_i)(x')\ \}\ \} \end{array} \quad =(5)$$

$\mathbf{tr}''$ is a member of a union if and only if it is a member of one of the sets considered:

$$\begin{array}{ll}\lambda r \lambda x.\{\ s(\overline{\mathbf{tr}''}.2(x)) \mid \mathbf{tr}'' \in \{ & [\ \mathbf{tr} \mid \langle r, \overline{\mathbf{tr}}.2\rangle\ ] \\ \mid & \mathbf{tr} \in \mathbf{trs}\ \wedge \\ & \forall x'\ .\ \{s(\overline{\mathbf{tr}'}.2(x'))|\mathbf{tr}' \in \mathbf{trs}\} \subseteq a(r_i)(x')\ \}\ \} \end{array} \quad =(5)$$

Since we only extract the most recent environment from the traces considered:

$$\lambda r \lambda x.\{\ s(\mathbf{env}_c(x)) \mid \mathbf{env}_c \in \mathbf{envs}_c\ \wedge\ \forall x'.\{s(\mathbf{env}'_c(x'))|\mathbf{env}'_c \in \mathbf{envs}_c\} \subseteq a(r_i)(x')\ \} \quad =(5)$$

We now define $\mathbf{env}_a$ as done in the previous cases:

$$\lambda r \lambda x.\{\ u \mid u \in \mathbf{env}_a(x)\ \wedge\ \forall x'.\mathbf{env}_a(x') \subseteq a(r_i)(x')\ \} \quad =(5)$$

And adding the union brings us to the correct definition:

$$\lambda r \lambda x. \bigcup_{i=1}^{n}\{\ \mathbf{env}_a(x) \mid \forall x'.\mathbf{env}_a(x') \subseteq a(r_i)(x')\ \} \quad \Box(5)$$

This shows that the monotone transfer function defined in 2.1.3 is equivalent to the one from 1.3.6, and since both computational machines start with the same initial state, the analyses are equivalent.

2.4 Safety and precision results

To show that the abstract interpretation is a safe approximation of the accumulating semantics, we can prove the following lemma, which says that a step in the accumulating semantics is always smaller than (and therefore less safe, but more precise than) a step in the analysis. This is a general result that will apply to any abstract interpretation defined in terms of α and γ.

Theorem 2.4.1. *Safety of analysis, where* $\boldsymbol{int} = \lambda c.\alpha(\boldsymbol{acc}(\gamma(c))$ *as proven above.*

$$\forall c.\mathbf{acc}(c) \sqsubseteq_c \gamma(\mathbf{int}(\alpha(c)))$$

Proof: definition 2.1.5 and equivalence of **int** with the abstract interpretation.

To show that the abstract interpretation is the most precise result, we show that there is no safe analysis that is strictly better than **int**. This proof comes from [6].

Theorem 2.4.2. ***int** is the most precise analysis:* Assume $f : A \rightarrow A$ is a safe monotone transfer function (i.e. $\forall c.\mathbf{acc}(c) \sqsubseteq_c \gamma(f(\alpha(c)))$, and it is "more precise than" **int**, i.e. $\exists a.f(a) \sqsubset_a \mathbf{int}(a)$. Then this leads to contradiction:

Proof:

We know that since f is safe:

$\forall c.\mathbf{acc}(c) \sqsubseteq_c \gamma(f(\alpha(c)))$

and by substitution of c and by theorem 2.2.5, we know that:

$\forall a.\alpha(\mathbf{acc}(\gamma(a)))) \sqsubseteq_a f(\alpha(\gamma(a)))$

and by definition 2.1.5, we know that:

$\forall a.\alpha(\mathbf{acc}(\gamma(a)))) \sqsubseteq_a f(a)$

and since this is the definition of **int**, this contradicts $\exists a.f(a) \sqsubset_a \mathbf{int}(a)$!

This concludes the study of abstract interpretation in its pure form. Next we augment an abstract interpretation with a widening operator in order to get better termination properties while retaining safety.

Chapter 3

Widening

Each analysis is a simple state machine, with an initial state in A, and a monotone transfer function $\mathbf{int} : A \rightarrow A$. Since the function is monotone and A is a lattice, we can imagine the analysis taking a path up the lattice, during the course of its many iterations. It will always ascend, since it can never descend due to its monotonicity, and if it remains still, the analysis will terminate and thus so will the path.

If the lattice is finite, then no matter what path is taken by the analysis, the path will always have finite length, and thus every analysis will terminate. The sign lattice in the sign analysis is finite, since the number of arcs in a program, the number of variables in a program, and the number of possible sets of signs are finite. If we had considered, rather than signs, an uncountable approximation of the set of values, then this would no longer be the case.

If we cannot guarantee an analysis will terminate, then it is not suitable for the static analysis of code such as used for compiler optimisations and error checking.

It is useful to have a method whereby a non terminating approximate analysis can be converted into an even more approximate terminating analysis. We want a flexible way of doing this while retaining safety. One simple idea that can be used here is to choose a point that is a finite distance from the base of the lattice, where upon we will jump right to the top of the lattice. This is analogous to entering a room containing an infinite number Computer Scientists, seeing that some finite number are male, and then terminating the analysis, concluding that there is the possibility that all of them are male.

This chapter summarises the theoretical machinery for implementing such ideas within any arbitrary analysis.

3.1 Widening operators

To capture the effect of artificially jumping forward during an analysis in a general way, we can think of a binary operator ($\overline{\nabla}$) which is inserted into the analysis's computational machine in the following way. We can think of the n^{th} step of the analysis being a_n and a_0 as the initial state of the analysis (the empty set of inferred properties). Inductively, the n^{th} state is $a_n = \mathbf{int}(a_{n-1})$. This is just a formalisation of the idea of a computing machine with an initial state and a state transfer function. We change this definition to introduce the widening operator like so:

Definition 3.1.1. *Analysis augmented with widening operator:*

$$(\overline{\nabla}) : (A \times A) \rightarrow A$$

$$a_n = \mathbf{int}(a_{n-1})\overline{\nabla} a_{n-1}$$

This means each new state is filtered through the widening operator, and the widening operator looks at both the current state, and the proposed next state, and can adjust the next state however it pleases. For this to be safe, the following must hold:

Definition 3.1.2. *Safe widening operators must satisfy:*

$$\forall a_1, a_2 \;.\; a_1 \overline{\nabla} a_2 \sqsupseteq_a a_1$$

The "accelerated" state must preserve any properties that were inferred by the analysis, and this requirement ensures this happens. The way that the operator is used ensures that $a_1 \sqsupseteq a_2$. Simply taking the widening operator to be the least upper bound operator produces the original analysis, so this technique generalises the abstract interpretation framework. It gifts us with the freedom to leap, where before, we could only cautiously tread.

The manner in which we can use this to force termination for infinite lattices is that we can use the widening operator to behave in the normal way until it reaches a certain point, then define it to return the top of the lattice. In general, we can make the widening operator have a *finite range.*

The approach taken in [2], [3] and [4] is to mark certain arcs to be "widening" arcs, such that each cycle in the flow chart has at least one widening arc. This is inspired by the knowledge that it is the loops in the program that create the problem of non-termination of the analysis, and by only treating these loops specially, we can minimise the damage done to the precision of the analysis.

It seems, that this does not make a huge amount of difference, since as soon as you artificially distort the state of an analysis at one arc, this distortion will propagate to the rest of the analysis as the flow of the program dictates. It will in some cases, produce a more accurate analysis, for instance if the program happens to step over the threshold after all its loops, on the way to its exit arc, but these situations are in the minority.

3.2 Interval Example

An analysis which abstracts the set of values held by a variable to the range in which they lie, is called an interval analysis. This analysis can be defined via abstract interpretation, but first a few auxiliary definitions that are largely intuitive:

Definition 3.2.1. *Notation for interval objects: (l ranges over intervals)*

Augmenting the set of values:	$v \in \mathbb{Z}^{\infty} = \mathbb{Z} \cup \{\infty, -\infty\}$
The set of all intervals:	$l \in \mathbf{Int} = (\mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty}) \cup \{[\emptyset, \emptyset]\}$
Constructing an interval:	$[v_1, v_2] \in \mathbf{Int}$
Constructing an "empty" interval:	$[\emptyset, \emptyset] \in \mathbf{Int}$
Decomposing an interval:	$\lfloor [v_1, v_2] \rfloor = v_1,\ \lceil [v_1, v_2] \rceil = v_2$ (not defined for empty intervals)
Least upper bound of integers:	$max : \mathbb{P}(\mathbb{Z}^{\infty}) \rightarrow (\mathbb{Z}^{\infty} \cup \emptyset)$
Greatest lower bound of integers:	$min : \mathbb{P}(\mathbb{Z}^{\infty}) \rightarrow (\mathbb{Z}^{\infty} \cup \emptyset)$
Interval comparison:	$l_1 \preccurlyeq l_2 \iff (\lfloor l_1 \rfloor \geq \lfloor l_2 \rfloor \bigwedge \lceil l_1 \rceil \leq \lceil l_2 \rceil) \bigvee l_1 = [\emptyset, \emptyset]$
Interval least upper bound:	$\curlyvee L' = [min\{\lfloor l \rfloor \mid l \in L'\}, max\{\lceil l \rceil \mid l \in L'\}]$
Interval greatest lower bound:	$\curlywedge L' = [max\{\lfloor l \rfloor \mid l \in L'\}, min\{\lceil l \rceil \mid l \in L'\}]$ (or $[\emptyset, \emptyset]$ if there is no overlap)

Definition 3.2.2. *The interval analysis abstraction and concretisation functions:*

$$\alpha(c) = \lambda r \lambda x.[\ \ min\{\overline{\mathbf{tr}}.2(x) | \mathbf{tr} \in c(r)\},\ \ max\{\overline{\mathbf{tr}}.2(x) | \mathbf{tr} \in c(r)\}\ \ ]$$
$$\gamma(a) = \lambda r.\{\ \mathbf{tr} \mid a(r)(x) \neq [\emptyset, \emptyset]\ \wedge\ \forall x\ .\ \lfloor a(r)(x) \rfloor \leq \overline{\mathbf{tr}}.2(x) \leq \lceil a(r)(x) \rceil\ \}$$

Definition 3.2.3. *Interval analysis using abstract interpretation:*

Set:	$A = \mathbf{Arc} \rightarrow \mathbf{Var} \rightarrow \mathbf{Int}$
Complete partial order:	$a_1 \sqsubseteq_a a_2 \iff \forall r, x\ .\ a_1(r)(x) \preccurlyeq a_2(r)(x)$
Least upper bound:	if $A' \subseteq A$, $\sqcup_a A' = \lambda r \lambda x. \curlyvee\{a(r)(x) | a \in A'\}$
Greatest lower bound:	if $A' \subseteq A$, $\sqcap_a A' = \lambda r \lambda x. \curlywedge\{a(r)(x) | a \in A'\}$
Top:	$\top_a = \lambda r.\lambda x.[-\infty, \infty]$
Bottom:	$\perp_a = \lambda r \lambda x.[\emptyset, \emptyset]$
Monotone transfer function:	$\lambda a.\alpha(\mathbf{acc}(\gamma(a)))$

This analysis has an infinite state, since **Int** is infinite, since $\mathbb{Z}^{\infty}$ is infinite. There is no guarantee that the analysis will terminate, and a trivial example of a program which causes the analysis to run indefinitely, is a simple infinite loop which constantly increments a variable, starting with 0.

In that situation, the interval abstraction for that variable will keep increasing in diameter, but will never reach $[0, \infty]$, which is clearly the only fixed point in can reach. We can define a widening operator to make the analysis "give up" after a certain amount of "fuel" has been used, and after that point it will return a fixed point and this will cause the analysis to stabilise and terminate.

Definition 3.2.4. *A widening operator for the interval analysis:* (the parameter $fuel$ is constant, and used to tune the analysis)

$$a_1 \overline{\nabla}^{fuel} a_2 = \lambda r \lambda x. \begin{cases} a_1(r)(x) & \text{if } a_1(r)(x) \preccurlyeq fuel \\ [-\infty, \infty] & \text{otherwise} \end{cases}$$

Note that this widening operator, when wrapped around the state transfer function as shown in definition 3.1.1, has a finite range (as long as the specified fuel is a finite interval). This means that no matter what strictly ascending path is taken through this range by the analysis, it must terminate in a finite number of steps.

The analysis, when applied to the example infinite loop program described above, will iterate until $fuel$ is reached, and then immediately jump to $[-\infty, \infty]$ and terminate. This technique is very costly in terms of accuracy since there is no way to investigate intervals greater than the limited specified. The limit is flexible however, and without losing safety, the compiler could tune it for the situation at hand.

This widening operator does not discriminate between arcs when performing the widening itself, so there is room for slightly more precision if a slightly more complex definition, that only widens the path of loops, is used. In this case the range of the iteration will not be finite in all cases, and we must show through other means that the iteration will always terminate.

There are many possible ways to use widening operators. In general we will have to prove that a widening operator really does ensure analysis computation sequences reach a fixed point in finite time. This proof will depend on what kind of widening operator is being used.

Chapter 4

Conclusion

We have described a simple language, motivated and defined its accumulating semantics, and suggested that this forms the foundation from which a large number of analyses can be derived. We described a sign analysis, and showed how it can be derived safely and completely from the accumulating semantics using mathematical notation. We have motivated the use of widening operators to place fixed upper limits on the number of iterations permitted in an analysis, and demonstrated how this gives an interval analysis guaranteed termination at considerable expense of precision.

This report is a summary and consolidation of what was thought to be the most important information from several sources. The chapter "abstract interpretation" in [7] contained a lot of theoretical results about Galois connections and widening, the most relevant of which has been presented here. The article [6] contains a very verbose introduction to abstract interpretation. It provided the context and intuition that [7] lacked.

The original papers [2], [3] and [4] provided the most concrete description of abstract interpretation, since they concentrated on a simple language and simple accumulating semantics. They also contained the most formal definition of the accumulating semantics out of the source material.

In this report, abstract interpretation has been considered with a very practical focus, and this limits the generality of the results presented. In order to present a fully worked example, the language used had to be very simple, and thus lacks many features considered necessary for conventional languages, such as dynamic binding and complex data structures. Considering these features in the context of abstract interpretation is non-trivial.

Despite this, the principle of abstract interpretation, i.e. the approximation of program semantics (using an accumulating semantics and a Galois insertion) is applicable in many languages and computational devices, from functional languages to hardware. Recent work has used abstract interpretation for static analysis of programs written in a cut-down version of the C language for the Aerospace industry [5]. Abstract interpretation is thus a powerful tool for static program analysis, since it combines safety with good termination properties, and manageable imprecision.

References

[1] Various Authors. Program analysis slides, 2001. Taught course at Imperial College, London.

[2] P. Cousot and R. Cousot. Static verification of dynamic type properties of variables. Res. rep. R.R. 25, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Nov. 1975. 18 p.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astree analyzer. In *ESOP'05*, 2005.

[6] N.D. Jones and F. Nielson. Abstract interpretation: a semantics based tool for program analysis. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 4*, pages 527–636. 1992.

[7] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.