Abstract Interpretation: a Semantics-Based Tool for Program Analysis

Neil D. Jones
DIKU, University of Copenhagen, Denmark

Flemming Nielson
Computer Science Department, Aarhus University, Denmark

June 30, 1994

Contents

1 Introduction

Desirable mathematical background for this chapter includes

- basic concepts such as lattices, complete partial orders, homomorphisms, etc.
- the elements of domain theory, e.g. as in the chapter by Abramsky or the books [Schmidt, 1986] or [Nielson, 1992a].
- the elements of denotational semantics, e.g. as in the chapter by Tennent or the books [Schmidt, 1986] or [Nielson, 1992a].
- interpretations as used in logic.

There will be some use of structural operational semantics [Kahn, 1987], [Plotkin, 1981], [Nielson, 1992a], for example deduction rules for a program's semantics and type system. The use of category theory will be kept to a minimum but would be a useful background for the domain-related parts of Section 3.

1.1 Goals and Motivations

Our primary goal is to obtain as much information as possible about a program's possible run time behaviour without actually having to run it on all input data; and to do this automatically. A widely used technique for such program analysis is nonstandard execution, which amounts to performing the program's computations using *value descriptions* or *abstract values* in place of the actual computed values. The results of the analysis must describe *all possible program executions*, in contrast to profiling and other run-time instrumentation which describe only one run at a time. We use the term "abstract interpretation" for a semantics-based version of nonstandard execution.

Nonstandard execution can be roughly described as follows:

- perform commands (or evaluate expressions, satisfy goals etc.) using stores, values, ... drawn from abstract value domains instead of the actual stores, values, ... used in computations.

- deduce information about the program's computations on actual input data from the resulting abstract descriptions of stores, values, ...

One reason for using abstract stores, values, ... instead of the actual ones is for computability: to ensure that analysis results are obtained in finite time. Another is to obtain results that describe the result of computations on a set of possible inputs. The "rule of signs" is a simple, familiar abstract interpretation using abstract values "positive", "negative" and "?" (the latter is needed to express, for example, the result of adding a positive and a negative number).

Another classical example is to check arithmetic computations by "casting out nines", a method using abstract values 0, 1,..., 8 to detect errors in hand computations. The idea is to perform a series of additions, subtractions and multiplications with the following twist: whenever a result exceeds 9, it is replaced by the sum of its digits (repeatedly if necessary). The result obtained this way should equal the sum modulo 9 of the digits of the result obtained by the standard arithmetic operations. For example consider the alleged calculation

$$123 * 457 + 76543 =?= 132654$$

This is checked by reducing 123 to 6, 457 to 7 and 76543 to 7, and then reducing $6 * 7$ to 42 and so further to 6, and finally $6 + 7$ is reduced to 4. This differs from 3, the sum modulo 9 of the digits of 132654, so the calculation was incorrect. That the method is correct follows from:

$$\begin{aligned}(10a \pm b) \bmod 9 &= (a \pm b) \bmod 9\\ a * b \pmod 9 &= (a \bmod 9 * b \bmod 9) \pmod 9\\ a + b \bmod 9 &= (a \bmod 9 + b \bmod 9) \pmod 9\end{aligned}$$

The method abstracts the actual computation by only recording values modulo 9. Even though much information is lost, useful results are still obtained since this implication holds: if the alleged answer modulo 9 differs from the answer got by casting out nines, there is definitely an error.

On the need for approximation Due to the unsolvability of the halting problem (and nearly any other question concerning program behaviour), no analysis that always terminates can be exact. Therefore we have only three alternatives:

- Consider systems with a finite number of finite behaviours (e.g. programs without loops) or decidable properties (e.g. type checking as in Pascal). Unfortunately, many interesting problems are not so expressible.

- Ask interactively for help in case of doubt. But experience has shown that users are often unable to infer useful conclusions from the myriads of esoteric facts provided by a machine. This is one reason why interactive program proving systems have turned out to be less useful in practice than hoped.
- Accept *approximate* but correct information.

Consequently most research in abstract interpretation has been concerned with effectively finding "safe" descriptions of program behaviour, yielding answers which, though sometimes too conservative in relation to the program's actual behaviour, never yield unreliable information. In a formal sense we seek a $\sqsubseteq$ relation instead of equality. The effect is that the price paid for exact computability is loss of precision.

A natural analogy: abstract interpretation is to formal semantics as numerical analysis is to mathematical analysis. Problems with no known analytic solution can be solved numerically, giving approximate solutions, for example a numerical result r and an error estimate ϵ. Such a result is *reliable* if it is certain that the correct result lies within the interval [r-ϵ, r+ϵ]. The solution is acceptable for practical usage if ϵ is small enough. In general more precision can be obtained at greater computational cost.

Safety Abstract interpretation usually deals with discrete non-numerical objects that require a different idea of approximation than the numerical analyst's. By analogy, the results produced by abstract interpretation of programs should be considered as correct by a pure semantician, as long as the answers are "safe" in the following sense. A boolean question can be answered "true", "false" or "I don't know", while answers for the rule of signs could be "positive", "negative" or "?". This apparently crude approach is analogous to the numerical analyst's, and for practical usage the problem is not to give uninformative answers too often, analogous to the problem of obtaining a small ϵ.

An approximate program analysis is *safe* if the results it gives can always be depended on. The results are allowed to be imprecise as long as they always err "on the safe side", so if boolean variable J is sometimes true, we allow it to be described as "I don't know", but not as "false". Again, in general more precision can be obtained at greater computational cost.

Defining the term "safe" is however a bit more subtle than it appears. In applications , e.g. code optimization in a compiler, it usually means "the result of abstract interpretation may safely be used for program transformation", i.e. without changing the program's semantics. To define safety it is essential to understand precisely how the abstract values are to be interpreted in relation to actual computations.

For an example suppose we have a function definition

$$f(X_1, \ldots, X_n) = exp$$

where exp is an expression in $X_1, \ldots, X_n$. Two subtly different *dependency analyses* associate with exp a subset of f's arguments:

Analysis I.

$$\{X_{i1}, \ldots, X_{im}\} = \{X_j \mid \text{exp's value depends on } X_j \text{ in } \textit{at least one} \text{ computation of } f(X_1, \ldots, X_n)\ \}$$

Analysis II.

$$\{X_{i1}, \ldots, X_{im}\} = \{X_j \mid \text{exp's value depends on } X_j \text{ in } \textit{every} \text{ computation of } f(X_1, \ldots, X_n)\}$$

For the example

$$f(W, X, Y, Z) = \textbf{if } W \textbf{ then } (X + Y) \textbf{ else } (X + Z)$$

analysis I yields {W, X, Y, Z}, which is the smallest variable set always sufficient to evaluate the expression. Analysis II yields {W, X}, signifying that regardless of the outcome of the test, evaluation of exp requires the values of both W and X, but not necessarily those of Y or Z.

These are both dependence analyses but have different modality. Analysis I, for possible dependence, is used in the binding time analysis phase of *partial evaluation*: a program transformation which performs as much as possible of a program's computation, when given knowledge of only some of its inputs. Any variable depending on at least one unknown input in at least one computation might be unknown at specialization time. Thus if any among W, X, Y, Z are unknown, then the value of exp will be unknown.

Analysis II, for definite dependence, is a *need analysis* identifying that the values of W and X will always be needed to return the value. Such analyses are used for to optimize program execution in lazy languages. The basis is that arguments definitely needed in a function call $f(e_1, e_2, e_3, e_4, e_5)$ may be pre-evaluated, e.g. using "call by value" for e_2 and e_3, instead of the more expensive "call by need".

Strictness. Finding needed variables involves tracing possible computation paths and variable usages. For mathematical convenience, many researchers work with a slightly weaker notion. A function is defined to be "strict" in variable A if whenever A's value is undefined, the value of exp will also be undefined, regardless of the other variables' values. Formally this means: if A has the undefined value $\perp$ then exp evaluates to $\perp$. Clearly f both needs and is strict in variables W and X in the example. For another, X is strict in a definition $f(X) = f(X) + 1$ since $f(\perp) = \perp$, even though it is not needed.

Violations of safety In practice unsafe data-flow analyses are sometimes used on purpose. For example, highly optimizing compilers may perform "code motion", where code that is invariant in a loop may be moved to a point just before the loop's entry. This yields quite substantial speedups for frequently iterated loops but it can also change termination properties: the moved code will be performed once if placed before the loop, even if the loop exit occurs without executing the body. Thus the transformed program could go into a nonterminating computation not possible before "optimization".

The decision as to whether such efficiency benefits outweigh problems of semantic differences can only be taken on pragmatic grounds. If one takes a "completely pure view" even using the associative law to rearrange expressions may fail on current computers.

We take a purist's view in this chapter, insisting on safe analyses and solid semantic foundations, and carefully defining the interpretation of the various abstract values we use.

Abstract interpretation cannot always be homomorphic A very well-established way to formulate the faithful simulation of one system by another is by a homomorphism from one algebra to another. Given two (one-sorted) algebras

$$(D, \{a_i : D^{k_i} \rightarrow D\}_{i \in I})$$

and

$$(E, \{b_i : E^{k_i} \rightarrow E\}_{i \in I})$$

with carriers D, E and operators a_i, b_i, a *homomorphism* is a function $\beta : D \rightarrow E$ such that for each i and $x_1, \ldots, x_{k_i} \in D$

$$\beta(a_i(x_1, \ldots, x_{k_i})) = b_i(\beta x_1, \ldots, \beta x_{k_i})$$

In the examples of sign analysis (to be given later) and casting out nines, abstract interpretation is done by a homomorphic simulation of the operations $+$, $-$ and $*$. Unfortunately, pure homomorphic simulation is not always sufficient for program analysis.

To examine the problem more closely, consider the example of nonrecursive imperative programs. The "state" of such a program might be a program control point, together with the values of all variables. The semantics is naturally given by defining a *state transition function*, for instance

$$\begin{aligned}
\text{State} &= \text{Program point} \times \text{Store} \\
\text{Store} &= \text{Variable} \rightarrow \text{Value} \\
\text{next} &: \text{State} \rightarrow \text{State}
\end{aligned}$$

where we omit formally specifying a language syntax and defining "next" on grounds of familiarity.

Consider the algebra (State, next : State $\rightarrow$ State) and an abstraction (AbState, **next** : AbState $\rightarrow$ AbState), where AbState is a set of abstract descriptions of states. A truly homomorphic simulation of the computation would be a function β : State $\rightarrow$ AbState such that the following diagram commutes:

$$\begin{tikzcd}
\text{State} \arrow[r, "\text{next}"] \arrow[d, "\beta"] & \text{State} \arrow[d, "\beta"] \\
\text{AbState} \arrow[r, "\textbf{next}"] & \text{AbState}
\end{tikzcd}$$

In this case β is a *representation function* mapping real states into their abstract descriptions, and **next** simulates next's effects, but is applied to abstract descriptions.

This elegant view is, alas, not quite adequate for program analysis. For an example, consider sign analysis of a program where

AbState = Program point $\times$ AbStore
AbStore = Variable $\rightarrow$ $\{+,-,\,?\}$
next : AbState $\rightarrow$ AbState

Representation function β preserves control points and maps each variable into its sign. (The use of abstract value "?", representing "unknown sign", will be illustrated later.) If the program contains

p : $Y := X + Y$; **goto** q

and the current state is (p, $[X \mapsto 1, Y \mapsto -2]$) then we have

$$\begin{aligned}
\beta(next((p,[X \mapsto 1, Y \mapsto -2]))) &= \beta((q,[X \mapsto 1, Y \mapsto -1])) \\
&= (q,[X \mapsto +, Y \mapsto -])
\end{aligned}$$

On the other hand the best that **next** can possibly do is:

$$\begin{aligned}\mathbf{next}(\beta((p,[X \mapsto 1, Y \mapsto -2]))) &= \mathbf{next}((p,[X \mapsto +, Y \mapsto -])) \\ &= (q,[X \mapsto +, Y \mapsto ?])\end{aligned}$$

since $X + Y$ can be either positive or negative, depending on the exact values of X, Y (unavailable in the argument of **next**). Thus the desired commutativity fails.

In general the best we can hope for is a *semihomomorphic* simulation. A simple way is to equip E with a partial order $\sqsubseteq$, where $x \sqsubseteq y$ intuitively means "x is a more precise description than y", e.g. $+ \sqsubseteq ?$.

In relation to *safe* value descriptions, as discussed in Section 1.1: if x is a safe description of precise value v, and $x \sqsubseteq y$, then we will also expect y to be a safe description of v.

Computations involving abstract values cannot be more precise than those involving actual values, so we weaken the homomorphism restriction by allowing the values computed abstractly to be less precise than the result of exact computation followed by abstraction.

We thus require that for each i and $x_1, \ldots, x_{k_i} \in D$

$$\beta(a_i(x_1, \ldots, x_{k_i})) \sqsubseteq b_i(\beta x_1, \ldots, \beta x_{k_i})$$

and that the operations b_i be monotone. For the imperative language this is described by:

$$\begin{tikzcd}
\text{State} \arrow[r, "\text{next}"] \arrow[dd, "\beta"] & \text{State} \arrow[d, "\beta"] \\
 & \rotatebox{-90}{$\sqsubseteq$} \\
\text{AbState} \arrow[r, "\mathbf{next}"] & \text{AbState}
\end{tikzcd}$$

The monotonicity condition implies

$$\beta(next^n(s)) \sqsubseteq \mathbf{next}^n(\beta(s))$$

for all states s and $n \geq 0$, so computations by sequences of state transitions are also safely modeled.

Abstract interpretation in effect simulates many computations at once A further complication is that "real world" execution steps cannot be simulated in a one-to-one manner in the "abstract world". In program fragment

$$\text{p: } \textbf{if } X > Y \textbf{ then goto } \text{q} \textbf{ else goto } \text{r}$$

next((p, $[X \mapsto +, Y \mapsto +]$)) could yield (q, $[X \mapsto +, Y \mapsto +]$) or (r, $[X \mapsto +, Y \mapsto +]$), since the approximate descriptions contain too little information to determine the outcome of the test. Operationally this amounts to *nondeterminism:* the argument to **next** does not uniquely determine its result. How is such nondeterminism in the abstract world to be treated?

One way is familiar from finite automata theory: we lift

$$\text{next} : \text{State} \rightarrow \text{State}$$

to work on *sets of states*, namely

$$\wp\text{next} : \wp(\text{State}) \rightarrow \wp(\text{State})$$

defined by

$$\wp\text{next (state-set)} = \{ \text{ next(s)} \mid s \in \text{state-set } \}$$

together with an abstraction function α: $\wp(\text{State}) \rightarrow \text{AbState}$. This direction, developed by Cousot and Cousot and described in section 2.2, allows **next** to remain a function.

Another approach is to let β be a relation instead of a function. This approach is described briefly in section 2.8 and is also used in section 3.

The essentially nondeterministic nature of abstract execution implies that abstract interpretation techniques may be used to analyse *nondeterministic programs* as well as deterministic ones. This idea is developed further in [Nielson, 1983].

1.2 Relation to Program Verification and Transformation

Program verification has similar goals to abstract interpretation. A major difference is that abstract interpretation emphasizes *approximate* program descriptions obtainable by *fully automatic* algorithms, whereas program verification uses deductive methods which can in principle yield more precise results, but are not guaranteed to terminate. Another difference is that an abstract interpretation, e.g. sign detection, must work uniformly for *all* programs in the language it is designed for. In contrast, traditional program verification requires one to devise a new set of statement invariants for every new program.

Abstract interpretation's major application is to determine the applicability or value of optimization and thus has similar goals to program transformation [Darlington, 1977]. However most program transformation as currently practiced still requires considerable human interaction and is so significantly less automatic than abstract interpretation. Further, program transformation often requires proofs that certain transformations can be validly applied; abstract interpretation gives one way to obtain these.

1.3 The Origins of Abstract Interpretation

The idea of computing by means of abstract values for analysis purposes is far from new. Peter Naur very early identified the idea and applied it in work on the Gier Algol compiler [Naur, 1965]. He coined the term *pseudo-evaluation* for what was later described as "a process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not on their values" [Jensen, 1991]. The same basic idea is found in [Reynolds, 1969] and [Sintzoff, 1972]. Sintzoff used it for proving a number of well-formedness aspects of programs in an imperative language, and for verifying termination properties.

These ideas were applied on a larger scale to highly optimizing compilers, often under the names program flow analysis or data-flow analysis [Hecht, 1977], [Aho, Sethi and Ullman, 1986], [Kam, 1976]. They can be used for extraction of more general program properties [Wegbreit, 1975] and have been used for many applications including: generating assertions for program verifiers [Cousot, 1977b], program validation [Fosdick, 1976] and [Foster, 1987], testing applicability of program transformations [Nielson, 1985a], compiler generation and partial evaluation [Jones, 1989], [Nielson, 1988b], estimating program running times [Rosendahl, 1989], and efficiently parallelizing sequential programs [Masdupuy, 1991,Mercouroff, 1991].

The first papers on automatic program analysis were rather ad hoc, and oriented almost entirely around one application: optimization of target or intermediate code by compilers. Prime importance was placed on efficiency, and the flow analysis algorithms used were not explicitly related to the semantics of the language being analysed. Signs of this can be seen in the well-known unreliability of the early highly optimizing compilers, indicating the need for firmer theoretical foundations.

1.4 A Sampling of Data-flow Analyses

We now list some program analyses that have been used for efficient implementation of programming languages. The aim is to show how large the spectrum of interesting program analyses is, and how much they differ from one another. Only a few of these have been given good semantic foundations, so the list could serve as a basis for future work. References include [Aho, Sethi and Ullman, 1986] and [Muchnick, 1981].

All concern analysing the subject program's behaviour at particular program points for optimization purposes. Following is a rough classification of the analyses, grouped according to the behavioural properties on which they depend:

Sets of values, stores or environments that can occur at a program point

Constant propagation finds out which assignments in a program yield constant values that can be computed at compile time.

Aliasing analysis identifies those sets of variables that may refer to the same memory cell.

Copy propagation finds those variables whose values equal those of other variables.

Destructive updating recognizes when a new binding of a value to a variable may safely overwrite the variable's previous value, e.g. to reduce the frequency of garbage collection in Lisp [Bloss and Hudak, 1985], [Jensen, 1991], [Mycroft, 1981], [Sestoft, 1989].

Groundness analysis (in logic programming) finds out which of a Prolog program's variables can only be instantiated to ground terms [Debray, 1986], [Søndergaard, 1986].

Sharing analysis (in logic programming) finds out which variable pairs can be instantiated to terms containing shared subterms [Debray, 1986], [Mellish, 1987], [Søndergaard, 1986].

Circularity analysis (in logic programming) finds out which unifications in Prolog can be safely performed without the time-consuming "occur check" [Plaisted, 1984], [Søndergaard, 1986].

Sequences of variable values

Variables invariant in loops identifies those variables in a loop that are assigned the same values every time the loop is executed; used in *code motion*, especially to optimize matrix algorithms.

Induction variables identifies loop variables whose values vary regularly each time the loop is executed, also to optimize matrix algorithms.

Computational past

Use-definition chains associates with a reference to X the set of all assignments $X := \ldots$ that assign values to X that can "reach" the reference (following the possible flow of program control).

Available expressions records the expressions whose values are implicitly available in the values of program variables or registers.

Computational future

Live variables variable X is *dead* at program point p if its value will never be needed after control reaches p, else *live*. Memory or registers holding dead variables may be used for other purposes.

Definition-use analysis associates with any assignment $X := \ldots$ the set of all places where the value assigned to X can be referenced.

Strictness analysis given a functional language with normal order semantics, the problem is to discover which parameters in a function call can be evaluated using call by value.

Miscellaneous

Mode analysis To find out which arguments of a Prolog "procedure" are *input*, i.e. will be instantiated when the procedure is entered, and which are *output*, i.e. will be instantiated as the result of calling the procedure [Mellish, 1987].

Interference analysis To find out which subsets of a of program's commands can be executed so that none in a subset changes variables used by others in the same set. Such sets are candidates for parallel execution on shared memory, vector or data flow machines.

1.5 Outline

Ideally an overview article such as this one should describe its area both in breadth and in depth - difficult goals to achieve simultaneously, given the amount of literature and number of different methods used in abstract interpretation. As a compromise section 2 emphasizes overview, breadth and connections with other research areas, while section 3 gives a more formal mathematical treatment of a domain-based approach to abstract interpretation using a two-level typed lambda calculus. (The motivation is that abstract interpretation of denotational language definitions allows approximation of a wide class of programming language properties.) Section 4 is again an overview, referencing some of the many abstract interpretations that have been seen in the literature. Section 5 contains a glossary briefly describing the many terms that have been introduced. Following is a more detailed overview.

Driven by examples, section 2 introduces several fundamental analysis concepts seen in the literature. The descriptions are informal, few theorems are proved, and some concepts are made more precise later within the framework of section 3.

The section begins with a list of program analyses used by compilers, and does a parity analysis of an example program. The shortcomings of naive analysis methods are pointed out, leading to the need for a more systematic framework. The framework used by Cousot for flow chart programs is introduced, using what we call the "accumulating" semantics, elsewhere the collecting or static semantics[1].

Appropriate machinery is introduced to approximate the accumulating semantics, and to prove the approximations safe. The distinction between independent attribute and relational analyses is made, and the latter are related to Dijkstra's predicate transformers. Backwards analyses are then briefly described.

It is then shown how domain-based generalizations of these ideas can be applied to languages defined by denotational semantics, thus going far beyond flow chart programs. The main tools used are interpretations and logical relations, and a general technique is introduced for proving safety.

Section 3 uses representation functions and logical relations, rather than abstraction of an accumulating semantics. The approach is *metalanguage* oriented and highly systematic, emphasizing the metalanguage for denotational definitions rather than particular semantic definitions of particular languages. It emphasizes compositionality with respect to domain constructors, and the extension from the approximation of basic values and functions to all the program's domains, analogous to the construction of a free algebra from a set of generators. The components of the following goal are precisely formulated:

$$\begin{array}{rcl} \text{abstract interpretation} & = & \text{correctness} \\ & + & \text{most precise analyses} \\ & + & \text{implementable analyses} \end{array}$$

[1]There is a terminological problem here: [Cousot, 1977a] used the term "static semantics", but this has other meanings, so several researchers have used the more descriptive "collecting semantics". Unfortunately this term too has been used in more than one way, so we have invented yet another term: "accumulating semantics".

Section 4 illustrates the need to interpret programs over domains other than abstractions of the accumulating semantics. Some program analyses not naturally expressed by abstracting either an accumulating or an *instrumented* semantics are exemplified, showing the need for more sophisticated analysis techniques, and an overview is given of some alternative approaches including tree grammars.

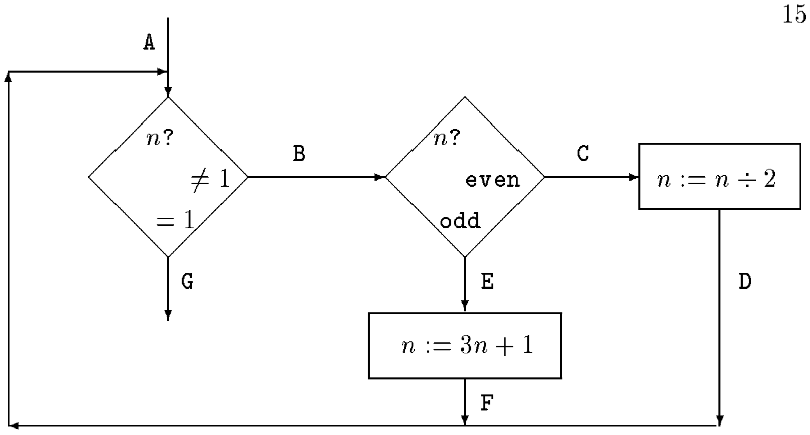
The idea of an "instrumented" semantics is introduced and correctness is discussed. This section is problem-oriented, with simulation techniques chosen ad hoc to fit the analysis problem and the language being analysed. It thus centers more around programs' operational behaviour than the structure of their domains, with particular attention to describing the set of program states reachable in computations on given input data, and to finite description of the set of all computations on given input. The section ends by describing approaches to abstract interpretation of Prolog.

2 Basic Concepts and Problems to be Solved

We begin with parity analysis of a very simple example program, and introduce basic concepts only as required. We discuss imperative programs without procedure calls since this familiar program class has a simple semantics and is most often treated in the analysis algorithms found in compiling textbooks. Later sections will discuss functional and logic programs, but many of their analysis problems are also visible, and usually in simpler form, in the imperative context. Throughout this section the reader is encouraged to ask himself "what is the analogue of this concept in a functional or logic programming framework?".

An example program, where ÷ stands for integer division (and program points A,...,G have been indicated for future reference):

```
A: while $n \neq 1$ do
   B: if $n$ even
        then (C: $n := n \div 2$; D: )
        else (E: $n := 3 * n + 1$; F: )
      fi
   od
G:
```



Side remark: Collatz' problem in number theory amounts to determining whether this program terminates for all positive initial n. To our knowledge it is still unsolved.

2.1 A Naive Analysis of the Simple Program

Abstraction of a single execution If this program is run with initial value $n = 5$, then n takes on values $5, 16, 8, 4, 2$ at point B, values $16, 8, 4, 2$ at C, etc. Using $\top$ to represent "either even or odd" the results of this single run can be abstracted as:

n at A	n at B	n at C	n at D	n at E	n at F	n at G
odd	$\top$	even	$\top$	odd	even	odd

Extension to all possible executions This result was obtained by performing *one* execution completely, and then abstracting its outcome. Such an analysis may of course not terminate, and it does not as wished describe all executions. The question is: how to obtain even-odd information valid for *all possible computations*? A natural way is to simulate the computation, but to do the computation using the abstract values

$$\text{Abs} = \{\bot, \text{even}, \text{odd}, \top\}$$

instead of natural numbers, each representing a set of possible values of n; and to ensure that all possible control flow paths are taken.

Doing this informally, we can see that if n is odd at program entry, it will always be even at points C and F, always odd at point E, sometimes even and sometimes odd at points B and D, and odd at G, provided control ever reaches G. Individual operations can be simulated by known properties of numbers, e.g. $3n + 1$ is even if n is odd and odd if n is even, while $n \div 2$ can be either even or odd.

Simulating the whole program is not as straightforward as simulating a single execution. The reason was mentioned before: execution over abstract values cannot in general be deterministic, since it must take account of all possible execution sequences on real data satisfying the abstract data description.

Towards a less naive analysis procedure The very earliest data-flow analysis algorithms amounted to glorified interpreters, and proceeded by executing the program symbolically, keeping a record of the desired flow information (abstract values) as the interpretation proceeded. Such algorithms, which in essence traced all possible control paths through the program, were very slow and often incorrect. They further suffered from a number of problems of semantic nature, for example difficulties in seeing how to handle nondeterminism due to tests with insufficient information to recognize their truth or falsity, convergence and divergence of control paths, loops and nontermination.

Better methods were soon developed to solve these problems, including

- putting a partial order on the abstract data values, so they always change in the same direction during abstract interpretation, thus reducing termination problems
- storing flow information in a separate data structure, usually bound to program points (such as entry points to "basic blocks", i.e. maximal linear program segments)
- constructing from the program a system of "data-flow equations", one for each program point
- solving the data-flow equations (usually by computing their greatest fixpoint or least fixpoint).

Much more efficient algorithms were developed and some theoretical frameworks were developed to make the new methods more precise; [Hecht, 1977], [Kennedy, 1981] and [Aho, Sethi and Ullman, 1986] contain good overviews.

None of the "classical" approaches to program analysis can, however, be said to be formally related to the semantics of the language whose programs were being analysed. Rather, they formalized and tightened up methods used in existing practice. In particular none of them was able to include *precise execution as a special case of abstract interpretation* (albeit an uncomputable one). This was first done in [Cousot, 1977a], the seminal paper relating abstract interpretation to program semantics.

2.2 Accumulating Semantics for Imperative Programs

The approach of [Cousot, 1977a] is appealing because of its generality: it expresses a large number of special program analyses in a common framework. In particular, this makes questions of safety (*i.e.* correctness) much easier to formulate and answer, and sets up a framework making it possible to relate and compare the precision of a range of different program analyses. It is solidly based in semantics, and precise execution of the program is included as a special case. This implies program verification may also be based on the accumulating semantics, a theme developed further in [Cousot, 1977b] and several subsequent works.

The ideas of [Cousot, 1977a] have had a considerable impact on later work in abstract interpretation, for example [Mycroft, 1981], [Muchnick, 1981], [Burn, 1986], [Donzeau-Gouge, 1978], [Nielson, 1982], [Nielson, 1984], [Mycroft, 1987]).

2.2.1 Overview of the Cousot Approach

The article [Cousot, 1977a] begins by presenting an operational semantics for a simple flow chart language. It then develops the concept of what we call the *accumulating semantics* (the same as Cousots' static semantics and some others' collecting semantics). This associates with each program point the set of all memory stores that can ever occur when program control reaches that point, as the program is run on data from a given initial data space. It was shown in [Cousot, 1977a] that a wide variety of flow analyses (but not all!) may be realized by finding finitely computable approximations to the accumulating semantics.

The (sticky) accumulating semantics maps program points to sets of program stores. The set $\wp(\text{Store})$ of all sets of stores forms a *lattice* with set inclusion $\subseteq$ as its partial order, so any two store sets A, B have least upper bound $A \cup B$ and greatest lower bound $A \cap B$. The lattice $\wp(\text{Store})$ is *complete*, meaning that any collection of sets of stores has a least upper bound in $\wp(\text{Store})$, namely its union.

Various approximations can be expressed by simpler lattices, connected to $\wp(\text{Store})$ by an *abstraction* function $\alpha : \wp(Store) \rightarrow \text{Abs}$ where Abs is a lattice of descriptions of sets of stores. Symbol $\sqcup$ is usually used for the least upper bound operation on Abs, $\sqcap$ for the greatest lower bound, and $\top, \bot$ for the least, resp. greatest elements of Abs.

An abstraction function is most often used together with a dual *concretization* function $\gamma : \text{Abs} \rightarrow \wp(\text{Store})$, and the two are required to satisfy natural conditions (given later).

For a one-variable program we could use as Abs the lattice with elements

$$\{\bot, \top, \text{even}, \text{odd}\},$$

where the abstraction of any nonempty set of even numbers is lattice element "even", and the concretization of lattice element "even" is the set of all even numbers. Abstract interpretation may thus be thought of as executing the program over a lattice of imprecise but computable *abstract store descriptions* instead of the precise and uncomputable accumulating semantics lattice.

In practice computability is often achieved by using a *noetherian* lattice, i.e. one without infinite ascending chains. More general lattices can, however, be used, cf. the Cousots' "widening" techniques, or the use of grammars to describe infinite sets finitely.

Let p_0 be the program's initial program point and let p be another program point. The set of store configurations that can be reached at program point p, starting from a set S_0 of possible initial stores is defined by:

$$\mathrm{acc}_p = \{s \mid (\mathrm{p},\, s) = \mathrm{next}^n((\mathrm{p}_0,\, s_0)) \text{ for some } s_0 \in S_0, n \geq 0\ \}$$

The accumulating semantics thus associates with each program point the set $\mathrm{acc}_p \subseteq$ Store.

2.2.2 Accumulating Semantics of the Example Program

For the example program there is only one variable, so a set of stores has form

$$\{[n \mapsto\ a_1], [n \mapsto\ a_2], [n \mapsto\ a_3], \ldots\}$$

For notational simplicity we can identify this with the set $\{\mathrm{a}_1,\, \mathrm{a}_2,\, \mathrm{a}_3,\, \ldots\}$ (an impossible simplification if the program has more than one variable). Given initial set $S_0 = \{5\}$ the sets of stores reachable at each program point are:

acc_A	acc_B	acc_C	acc_D	acc_E	acc_F	acc_G
{5}	{5,16,8,4,2}	{16,8,4,2}	{8,4,2,1}	{5}	{16}	{1}

The following *data-flow* equations have a unique least fixpoint by completeness of $\wp$(Store), and it is easy to see that their fixpoint is exactly the tuple of sets of reachable stores as defined above.

$$\begin{array}{lcl}
\mathrm{acc}_A & = & S_0 \\
\mathrm{acc}_B & = & (\mathrm{acc}_A \cup\ \mathrm{acc}_D \cup\ \mathrm{acc}_F) \cap \{n \mid\ n \in \{0,1,2,\ldots\} \setminus \{1\}\} \\
\mathrm{acc}_C & = & \mathrm{acc}_B\ \cap\ \{n \mid n \in \{0,2,4,\ldots\}\} \\
\mathrm{acc}_D & = & \{n \div 2 \mid n \in \mathrm{acc}_C\} \\
\mathrm{acc}_E & = & \mathrm{acc}_B \cap\ \{n \mid n \in \{1,3,5,\ldots\}\} \\
\mathrm{acc}_F & = & \{3n+1 \mid n \in \mathrm{acc}_E\}
\end{array}$$

$$\mathrm{acc}_G \quad = \quad (\mathrm{acc}_A \,\cup\, \mathrm{acc}_D \,\cup\, \mathrm{acc}_F) \cap \{1\}$$

The equation set can be derived mechanically from the given program's syntax, e.g. as seen in [Cousot, 1977a] or [Nielson, 1982].

2.2.3 Abstract Interpretation of the Example Program

The *abstraction function* $\alpha : \wp(\mathrm{Store}) \to \mathrm{Abs}$ below may be used to abstract a set of stores, where $\mathrm{Abs} = \{\bot, \text{even}, \text{odd}, \top\}$:

$$\alpha(S) = \begin{cases} \bot & \text{if S} = \{\}, \text{ else} \\ \text{even} & \text{if S} \subseteq \{0, 2, 4, \ldots\}, \text{ else} \\ \text{odd} & \text{if S} \subseteq \{1, 3, 5, \ldots\}, \text{ else} \\ \top & \end{cases}$$

Defining $\bot \sqsubseteq \text{even} \sqsubseteq \top$ and $\bot \sqsubseteq \text{odd} \sqsubseteq \top$ makes Abs into a partially ordered set. Least upper and greatest lower bounds $\sqcup, \sqcap$ exist so it is also a lattice.

$$\begin{tikzcd} & \top \arrow[dl, dash] \arrow[dr, dash] & \\ \text{even} \arrow[dr, dash] & & \text{odd} \arrow[dl, dash] \\ & \bot & \end{tikzcd}$$

Applying α to the sets of reachable stores yields the following:

abs_A	abs_B	abs_C	abs_D	abs_E	abs_F	abs_G
odd	$\top$	even	$\top$	odd	even	odd

Abstraction of the set of all runs This method is still unsatisfactory for describing all computations since the value sets involved are unbounded and possibly infinite. But we may model the equations above by applying α to the sets involved. The abstraction function α just given is easily seen to be monotone, so set inclusion $\subseteq$ in the world of actual computations is modelled by $\sqsubseteq$ in the world of simulated computations over Abs. Union is the least upper bound over sets, so it is natural to model $\cup$ by $\sqcup$, and similarly to model $\cap$ by $\sqcap$.

The arithmetic operations are faithfully modelled as follows, using familiar properties of natural numbers:

$$f_{n\div 2}(abs) = \begin{cases} \bot & \text{if } abs = \bot \\ \top & \text{else} \end{cases}$$

$$f_{3n+1}(abs) = \begin{cases} \bot & \text{if } abs = \bot\text{, else} \\ \text{even} & \text{if } abs = \text{odd, else} \\ \text{odd} & \text{if } abs = \text{even, else} \\ \top & \text{if } abs = \top \end{cases}$$

This yields the following system of *approximate data-flow equations*, describing the program's behaviour on Abs:

$$\begin{array}{lcl} abs_A & = & \alpha(S_0) \\ abs_B & = & (abs_A \sqcup abs_D \sqcup abs_F) \sqcap \top \quad (\text{“}\sqcap\top\text{” may be omitted}) \\ abs_C & = & abs_B \sqcap \text{even} \\ abs_D & = & f_{n\div 2}(abs_E) \\ abs_E & = & abs_B \sqcap \text{odd} \\ abs_F & = & f_{3n+1}(abs_E) \\ abs_G & = & (abs_A \sqcup abs_D \sqcup abs_F) \sqcap \text{odd} \end{array}$$

Remark Here $f_{n\div 2}$ and f_{3n+1} were defined ad hoc; a systematic way to define them will be seen in section 2.3.

The lattice Abs is also complete. The operators $\sqcap$, $\sqcup$, $f_{n\div 2}$ and f_{3n+1} are monotone, so the equation system has a (unique) least fixpoint. The abstraction function α is easily seen to be monotone, so if it also were a homomorphism with respect to $\cup$, $\sqcup$ and $\cap$, $\sqcap$, the least solution to the approximate flow equations would be exactly

$$abs_A = \alpha(\text{acc}_A), \ldots, abs_G = \alpha(\text{acc}_G).$$

It is, however, *not* homomorphic since for example

$$\alpha(\{2\}) \sqcap \alpha(\{4\}) = \text{even} \neq \bot = \alpha(\{2\} \cap \{4\})$$

On the other hand the following *do* hold:

$$\begin{array}{rclrcl}
\alpha(A)\sqcup\alpha(B) & = & \alpha(A\cup B) & f_{n\div 2}(\alpha(A)) & \sqsupseteq & \alpha(\{n\div 2 \mid n\in A\}) \\
\alpha(A)\sqcap\alpha(B) & \sqsupseteq & \alpha(A\cap B) & f_{3n+1}(\alpha(A)) & \sqsupseteq & \alpha(\{3n+1 \mid n\in A\})
\end{array}$$

Using these, it is easy to see by inspection of the two equation systems (more formally: a simple fixpoint induction) that their least fix points are related by:

$$\begin{array}{rcl}
abs_A & \sqsupseteq & \alpha(\mathrm{acc}_A), \\
abs_B & \sqsupseteq & \alpha(\mathrm{acc}_B), \\
 & \vdots & \\
abs_G & \sqsupseteq & \alpha(\mathrm{acc}_G)
\end{array}$$

Following is the iterative computation of the least fixpoint, assuming $S_0 = \{5\}$:

abs_A	abs_B	abs_C	abs_D	abs_E	abs_F	abs_G	iteration
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	0
odd	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	1
odd	odd	$\perp$	$\perp$	$\perp$	$\perp$	odd	2
odd	odd	$\perp$	$\perp$	odd	$\perp$	odd	3
odd	odd	$\perp$	$\perp$	odd	even	odd	4
odd	$\top$	$\perp$	$\perp$	odd	even	odd	5
odd	$\top$	even	$\perp$	odd	even	odd	6
odd	$\top$	even	$\top$	odd	even	odd	7, 8, ...

The conclusion is that n is always even at points C and F, and always odd at E and G.

2.2.4 An Optimization Using the Results of the Analysis

The flow analysis reveals that the program could be made somewhat more efficient by "unrolling" the loop after F. The reason is that tests "$n \neq 1$" and "n even" must be both be true in the iteration after F, so they need not be performed. The result is

```
while $n \neq 1$ do if $n$ even then $n := n \div 2$ else $n := (3*n+1)\div 2$
fi od
```

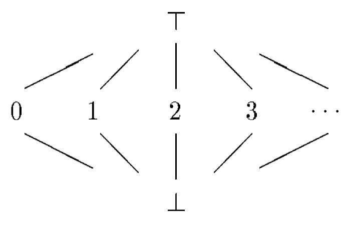
which avoids the two tests every time n is odd. In practice, one of the main reasons for doing abstract interpretation is to find out when such optimizing transformations may be performed.

2.2.5 Termination

The least fixed point may (as usual) be computed by beginning with $[\mathrm{pp}_1 \mapsto \perp, \ldots, \mathrm{pp}_m \mapsto \perp]$ (every program point is mapped to the least element of Abs), and repeatedly replacing the value currently assigned to pp_i by the value of the right side of pp_i's equation. By monotonicity of $\sqcap$, $f_{n\div 2}$ etc., these values can only grow or remain unchanged, so the iterations terminate provided the approximation lattice has no ascending chains of infinite height, as is the case here.

[Cousot, 1977a] describes ways to achieve termination even when infinite chains exist, by inserting so-called *widening* operators in the data-flow equations at each junction point of a loop. To explain the basic idea consider the problem of finding the fixed point of a continuous function f. The usual Kleene iteration sequence is $d_0 = \perp, \cdots, d_{n+1} = f(d_n), \cdots$ and is known to converge to the least fixed point of f but the sequence need not stabilize, i.e. it need not be the case that $d_{n+1} = d_n$ for some n. To remedy this one may introduce a *widening* operator $\bigtriangledown$ that dominates the least upper bound operation, i.e. $d' \sqcup d'' \sqsubseteq d' \bigtriangledown d''$, and such that the chain $d_0 = \perp, \cdots, d_{n+1} = d_n \bigtriangledown f(d_n)$ always stabilizes. This leads to overshooting the least fixed point but always gives a safe solution. By iterating down from the stabilization-value (perhaps by using the technique of *narrowing*) one may then be able to recover some of the information lost.

Constant propagation This is an example of a lattice which is infinite but has finite height (three). It is used for detecting variables that don't vary, and has Abs $= \{\top, \perp, 0, 1, 2, \ldots\}$ where $\perp \sqsubseteq n \sqsubseteq \top$ for $n = 0, 1, 2, \ldots$.



The corresponding abstraction function is:

$$\alpha(V) = \begin{cases} \perp & \text{if V} = \{\ \} \\ n & \text{if V} = \{n\} \\ \top & \text{otherwise} \end{cases}$$

There also exist lattices in which all ascending chains have finite height, even though the lattice as a whole has unbounded vertical extent. An example: let Abs $= (N, \geq)$.

2.2.6 Safety: First Discussion

The analysis of the Collatz-sequence program is clearly "safe" in the following sense: if control reaches point C then the value of n will be even, and similarly for the other program points and abstract values. Correctness (or soundness) of the even-odd analysis for *all possible* programs and program points is also fairly easy to establish, given the close connection of the flow equations to those defining the accumulating semantics.

Reachable program points A similar but simpler *reachability* analysis (e.g. for dead code elimination) serves to illustrate a point concerning safety. It uses Abs = $\{\top, \perp\}$ with $\perp \sqsubseteq \top$ and abstraction function α defined as follows (where $a \in$ Abs and $S \subseteq$ Store):

$$\begin{array}{lclll}
\alpha(S) & = & \perp & \text{if } S = \{\} & \text{else } \top \\
f_{n\div 2}(a) & = & \perp & \text{if } a = \perp & \text{else } \top \\
f_{3n+1}(a) & = & \perp & \text{if } a = \perp & \text{else } \top
\end{array}$$

Intuitively, $\perp$ abstracts only the empty set of stores and so appropriately describes unreachable program points, while $\top$ describes reachable program points. Computing the fixpoint as above we get:

abs_A	abs_B	abs_C	abs_D	abs_E	abs_F	abs_G
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

This might be thought to imply that *all* program points including G are reachable, regardless of the initial value of n. On the other hand, reachability of G for input n implies termination, and it is a well-known open question whether the program does in fact terminate for all n.

A more careful analysis reveals that $\perp$ at program point p represents "p *cannot* be reached", while $\top$ represents "p *might* be reached" and so does not necessarily imply termination. The example shows that we must examine the questions of correctness and safety more carefully, which we now proceed to do.

2.3 Correctness and Safety

In this and remaining parts of section 2, we describe informally several different approaches to formulating safety and correctness, and discuss some advantages and disadvantages. A more detailed domain-based framework will be set up in section 3.

2.3.1 Desirable Properties of the Abstract Value Set Abs

In order to model the accumulating semantics equations, Abs could be a *complete lattice*: a set with a partial order $\sqsubseteq$, with least upper and greatest lower bounds $\sqcup$ and $\sqcap$ to model $\cup$ and $\cap$, and such that any collection of sets of stores has a least upper bound in Abs. Note: any lattice of finite height is complete. In the following we sometimes write $a \sqsupseteq a'$ in place of $a' \sqsubseteq a$.

2.3.2 Desirable Properties of the Abstraction Function

Intuitively "even" represents the set of all even numbers. This viewpoint is made explicit in [Cousot, 1977a] by relating complete lattices Conc and Abs to each other by a pair α, γ of *abstraction and concretization* functions with types

$$\begin{array}{rlcl} \alpha : & \text{Conc} & \rightarrow & \text{Abs} \\ \gamma : & \text{Abs} & \rightarrow & \text{Conc} \end{array}$$

In the even-odd example above the lattice of concrete values is Conc = $\wp$(Store), and the natural concretization function is

$$\begin{array}{rcl} \gamma(\bot) & = & \{\} \\ \gamma(\mathit{even}) & = & \{0, 2, 4, \ldots\} \\ \gamma(\mathit{odd}) & = & \{1, 3, 5, \ldots\} \\ \gamma(\top) & = & \{0, 1, 2, 3, \ldots\} = \mathbb{N} \end{array}$$

Cousot and Cousot impose natural conditions on α and γ (satisfied by the examples):

1. α and γ are monotonic
2. $\forall a \in \text{Abs}$, $a = \alpha(\gamma(a))$
3. $\forall c \in \text{Conc}$, $c \sqsubseteq_{Conc} \gamma(\alpha(c))$

For the accumulating semantics, larger abstract values represent larger sets of stores by condition 1. Condition 2 is natural, and condition 3 says that $S \subseteq \gamma(\alpha(S))$ for any $S \subseteq$ Store.

The conditions can be summed up as: (α, γ) form a Galois insertion of Abs into $\wp$(Store), a special case of an adjunction in the sense of category theory. It is easy to verify the following

Lemma 1 If conditions 1-3 hold, then

- $\forall c \in$ Conc, a $\in$ Abs: $c \sqsubseteq_{Conc} \gamma(a)$ if and only if $\alpha(c) \sqsubseteq_{Abs} a$, and

- α is continuous

$\square$

Thus the abstract flow equations converge to a fixpoint. If α is semihomomorphic on union, intersection and base functions, then the abstract flow equations' fixpoint will be pointwise larger than or equal to the abstraction of the fixpoint of the accumulating semantics' equations.

Again, note that stores are unordered, so α and γ need only preserve the subset ordering. The more complex situation that arises when modelling nonflat domains is investigated in [Mycroft, 1983].

2.3.3 Safety: Second Discussion

Recalling the program of section 2.2, we can define the solution $(\text{abs}_A, \ldots, \text{abs}_G) \in \text{Abs}^7$ to the abstract flow equations to be *safe* with respect to the accumulating semantics $(\text{acc}_A, \ldots, \text{acc}_G) \in \wp(\text{Store})^7$ if the reachable sets of stores are represented by the abstract values:

$$\begin{aligned} acc_A &\subseteq \gamma(abs_A), \\ acc_B &\subseteq \gamma(abs_B), \\ &\vdots \\ acc_G &\subseteq \gamma(abs_G) \end{aligned}$$

This is easy to verify for the even-odd abstraction given before.

Returning to the question raised after the "reachable program points" example, we see that safety at point G only requires that $\text{acc}_G \subseteq \gamma(\text{abs}_G)$, i.e. that every store that can reach G appears in $\gamma(\text{abs}_G)$. This also holds if acc_G is empty, so $\gamma(\text{abs}_G) = \top$ does *not* imply that G is reachable in any actual computation. For any program point X, $\text{abs}_X = \bot$ implies $\text{acc}_X = \{\}$, which signifies that control cannot reach X. Thus abstract value $\bot$ can be used to eliminate dead code.

Safe approximation of base functions Consider a base function $\text{op} : \mathbb{N} \rightarrow \mathbb{N}$, and extend it, by "pointwise lifting" to sets of numbers, yielding $\wp\text{op} : \wp(\mathbb{N}) \rightarrow \wp(\mathbb{N})$ where

$$\wp\text{op}(N) = \{\text{op}(n) \mid n \in N\}$$

Suppose α, γ satisfy conditions 1-3. It is natural to define $\textbf{op} : \text{Abs} \rightarrow \text{Abs}$ to be a *safe approximation* to op if the following holds for all $N \subseteq \mathbb{N}$:

$$\wp\text{op}(N) \subseteq \gamma(\textbf{op}(\alpha(N)))$$

or, diagrammatically:

$$\begin{tikzcd}
\wp(\mathbb{N}) \arrow[r, "\wp\text{op}"] \arrow[dd, "\alpha"] & \wp(\mathbb{N}) \\
 & \rotatebox{90}{$\supseteq$} \\
\text{Abs} \arrow[r, "\mathbf{op}"] & \text{Abs} \arrow[u, "\gamma"']
\end{tikzcd}$$

By the conditions and lemma this is equivalent to

$$\alpha(\wp\text{op}(N)) \sqsubseteq \mathbf{op}(\alpha(\mathbf{N}))$$

corresponding to diagram:

$$\begin{tikzcd}
\wp(\mathbb{N}) \arrow[r, "\wp\text{op}"] \arrow[dd, "\alpha"] & \wp(\mathbb{N}) \arrow[d, "\alpha"] \\
 & \rotatebox{90}{$\sqsupseteq$} \\
\text{Abs} \arrow[r, "\mathbf{op}"] & \text{Abs}
\end{tikzcd}$$

Intuitively, for any subset $N \subseteq \mathbb{N}$, applying the induced abstract operation **op** to the abstraction of N represents at least all the values obtainable by applying op to members of N.

Induced approximations to base functions We now show how the best possible approximation **op** can be extracted from op (at least in principle, although perhaps not computably; a more detailed discussion appears in section 3.4). Recall that smaller elements of Abs abstract smaller sets of concrete values and so are less approximate, i.e. more precise descriptions.

Lemma 2 Given $\alpha : \wp(\mathbb{N}) \to \text{Abs}$ and $\gamma : \text{Abs} \to \wp(\mathbb{N})$ satisfying the three conditions above, define the operator *induced* by op to be $\mathbf{op} : \text{Abs} \to \text{Abs}$ where

$$\mathbf{op} = \alpha \circ \wp\text{op} \circ \gamma$$

Then $\mathbf{op}$ is the most precise function on Abs satisfying $\alpha(\wp\text{op}(N)) \sqsubseteq \mathbf{op}(\alpha(N))$ for all N.

Proof Suppose $f : \text{Abs} \to \text{Abs}$ with $\alpha(\wp\ \text{op}(N)) \sqsubseteq f(\alpha(N))$ for all N. Then for any a,

$$\mathbf{op}(a) = \alpha(\wp\text{op}(\gamma(a))) \sqsubseteq f(\alpha(\gamma(a))) = f(a)$$

□

The definition of $\mathbf{op}$ as a diagram:

$$\begin{tikzcd} \wp(\mathbb{N}) \arrow[r, "\wp\text{op}"] & \wp(\mathbb{N}) \arrow[d, "\alpha"] \\ \text{Abs} \arrow[u, "\gamma"] \arrow[r, "\mathbf{op}"] & \text{Abs} \end{tikzcd}$$

For example, if $\text{op}(n) = n \div 2$ then $\mathbf{op}$ is $f_{n \div 2}$ as seen above, e.g.

$$\begin{array}{llll} \mathbf{op}(\bot) & = \alpha(\{n \div 2 \mid n \in \gamma(\bot)\}) & = \alpha(\{\}) & = \bot \\ \mathbf{op}(\text{even}) & = \alpha(\{n \div 2 \mid n \in \gamma(\text{even}\}) & = \alpha(\{0, 1, 2, \ldots\}) & = \top \end{array}$$

Unfortunately the definition of $\mathbf{op}$ does not necessarily give a terminating algorithm for computing it, even if op is computable. In practice the problem is solved by approximating from above, i.e. choosing $\mathbf{op}$ to give values in Abs that may be larger (less informative) than implied by the above equation. We will go deeper into this in Subsection 3.5.

A local condition for safe approximation of transitions A safety condition on one-step transitions can be formulated analogously. Define for any two control points p, q the function $\text{next}_{p,q}:\wp(\text{Store}) \rightarrow \wp(\text{Store})$:

$$\text{next}_{p,q}(S) = \{s' \mid (q, s') = \text{next}((p, s)) \text{ for some } s \in S\}$$

This is the earlier transition function, extended to include all transitions from p to q on a set of stores. Exactly as above we can define the abstract transition function *induced* by α and γ to be

$$\mathbf{next}_{p,q} = \alpha \circ \text{next}_{p,q} \circ \gamma$$

This is again the most precise function satisfying

$$\mathbf{next}_{p,q}(\alpha(S)) \sqsubseteq \alpha(\text{next}_{p,q}(S))$$

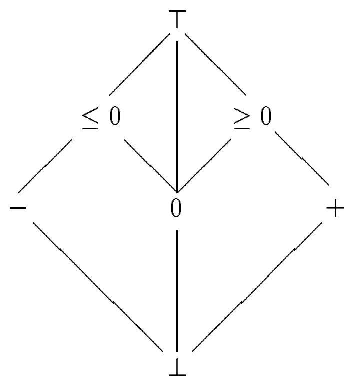
for all S.

2.3.4 An Example: the Rule of Signs

Consider the abstract values +, - and 0 with the natural concretization function

$$\begin{array}{rcl} \gamma(0) & = & \{0\} \\ \gamma(+) & = & \{1, 2, 3, ...\} \\ \gamma(-) & = & \{-1, -2, -3, \ldots\} \end{array}$$

This can be made into a complete lattice by adding greatest lower and least upper bounds in various ways. Assuming $\sqcap$, $\sqcup$ should model $\cap$, $\cup$ respectively, the following is obtained:



with

$$\begin{aligned}\gamma(\bot) &= \{\}\\ \gamma(\geq 0) &= \{0,1,2,3,...\}\\ \gamma(\leq 0) &= \{0,-1,-2,-3,...\}\\ \gamma(\top) &= \{\ldots,-2,-1,0,1,2,3,\ldots\} = \mathbb{Z}\end{aligned}$$

and abstraction function

$$\alpha(S) = \begin{cases} \bot & \text{if } S = \{\} \text{ else}\\ + & \text{if } S \subseteq \{1,\ 2,\ 3,...\} \text{ else}\\ \geq 0 & \text{if } S \subseteq \{0,\ 1,\ 2,\ 3,\ldots\} \text{ else}\\ - & \text{if } S \subseteq \{\text{-}1,\text{-}\ 2,\ \text{-}3,\ldots\} \text{ else}\\ \leq 0 & \text{if } S \subseteq \{0,\ \text{-}1,\ \text{-}2,\ \text{-}3,\ldots\} \text{ else}\\ \top & \end{cases}$$

The induced approximation for operator $+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ is:

$+'$	$\bot$	$-$	0	$+$	≥ 0	≤ 0	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$-$	$\top$	$\top$	$-$	$\top$
0	$\bot$	$-$	0	$+$	≥ 0	≤ 0	$\top$
$+$	$\bot$	$\top$	$+$	$+$	$+$	$\top$	$\top$
≥ 0	$\bot$	$\top$	≥ 0	$+$	≥ 0	$\top$	$\top$
≤ 0	$\bot$	$-$	≤ 0	$\top$	$\top$	≤ 0	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

2.3.5 Composition of Safety Diagrams

Suppose we have two diagrams for safe approximation of two base functions op and op':

$$\begin{tikzcd}
\wp(\mathbb{N}) \arrow[r, "\wp\mathrm{op}"] \arrow[d, "\alpha"] & \wp(\mathbb{N}) \arrow[d, "\alpha"] \arrow[d, "\sqcap|"' near end] \\
\mathrm{Abs} \arrow[r, "\mathbf{op}"] & \mathrm{Abs}
\end{tikzcd}$$

and

$$\begin{tikzcd}
\wp(\mathbb{N}) \arrow[r, "\wp\text{op}'"] \arrow[d, "\alpha"] & \wp(\mathbb{N}) \arrow[d, "\alpha"] \\
\text{Abs} \arrow[r, "\mathbf{op}'"] & \sqsupseteq \text{Abs}
\end{tikzcd}$$

It is easy to see that $\mathbf{op}'\circ\,\mathbf{op}$ is a safe approximation to $\wp$ op$'\circ\wp$ op, so the two may be composed:

$$\begin{tikzcd}
\wp(\mathbb{N}) \arrow[r, "\wp\text{op}'\circ\ \wp\text{op}"] \arrow[d, "\alpha"] & \wp(\mathbb{N}) \arrow[d, "\alpha"] \\
\text{Abs} \arrow[r, "\mathbf{op}'\circ\,\mathbf{op}"] & \sqsupseteq \text{Abs}
\end{tikzcd}$$

On the other hand the diagrams for the *induced* approximations to base functions *cannot* be so composed, since the best approximation to $\wp$op$'\circ\wp$op may be better than the composition of the best approximations to $\wp$op and $\wp$op$'$. (This is precisely because α is a semihomomorphism, not a homomorphism.) For a concrete example, let op and op$'$ respectively describe the effects of the two assignments

$$n := 4 * n + 2;\ \ n := n \div 2$$

Then

$$\alpha(\wp\text{op}'\circ\wp\text{op}(\{0,1,2,...\})) = \alpha(\{1,3,5,...\}) = \mathit{odd}$$

whereas

$$\mathbf{op'}\circ\,\mathbf{op}\ (\alpha(\{0,1,2,...\}) = \mathbf{op'}(\text{even}) = \top.$$

2.4 Scott Domains, Lattice Duality, and Meet versus Join

Relation to Scott-style domains The partial order $\sqsubseteq$ on Abs models the set inclusion order $\subseteq$ used for $\wp$(Store) in the accumulating semantics. In abstract interpretation, larger elements of Abs correspond to *more approximate* descriptions, so if $a \sqsubseteq a'$ then a' describes a *larger set* of concrete values. For example, "even" describes any set of even numbers, and $\top$ describes the set of all numbers.

In contrast, Scott domains as used in denotational semantics use an ordering by "information content", where a larger domain element describes *a single value that is more completely calculated.* During a computation $\bot$ means "not yet calculated", intuitively a slot to be filled later in with the final value. Appearance of $\bot$ in a program's final result signifies "was never filled in", and so represents nontermination (at least in languages with eager evaluation).

A value in a Scott domain represents perhaps incomplete knowledge about a *single* program value, for example a finite part of an infinite function f. The partial order $f \sqsubseteq f'$ signifies that f' is more completely defined than f, and that f' agrees with f where ever it is defined. $\top$, if used at all, indicates inconsistent values.

Clearly this order is not the same as the one used in abstract interpretation, and the difference is more than just one of duality.

Least or Greatest Fixpoints? Literature on data-flow analysis as used in compilers [Aho, Sethi and Ullman, 1986,Hecht, 1977,Kennedy, 1981] often uses abstract value lattices which are dual to the ones we consider, so larger elements represent more precise descriptions rather than more approximate. This is mainly a matter of taste; but has the consequence that *greatest* fixpoints are computed instead of least ones, and that the $\cup$ and $\cap$ of the accumulating semantics are modelled by $\sqcap$ and $\sqcup$, respectively. We prefer least fixpoints due to their similarity to those naturally used in defining the accumulating semantics.

Should $\sqcup$ or $\sqcap$ be Used on Converging Paths? We have argued that $\sqcup$ naturally models the effect of path convergence because it corresponds to $\cup$ in the accumulating semantics. On the other hand, there exist abstract interpretations that are *not* approximations to the accumulating semantics, and for some of these path convergence is properly modelled by $\sqcap$. To see this, consider the two dependence analyses mentioned in section 1.1. For analysis I, path convergence should be modelled by $\sqcup$ since a variable dependence is to be recorded if it occurs along *at least one* path. For analysis II it should be modelled by $\sqcap$ since a dependence is recorded only if it occurs along *all* paths. So the choice between $\sqcup$ and $\sqcap$ on converging paths is just another incarnation of the modality distinction encountered in section 1.

2.5 Abstract Values Viewed as Relations or Predicates

The accumulating semantics binds to each program point a set of stores. Suppose the program's variables are $V_1,\ldots,V_n$, so a store is an element of Store = $\{V_1,\ldots,V_n\} \to$ Value. In the examples above there was only one variable, so a set of stores was essentially a set of values, which simplified the discussion considerably. The question arises: how can we abstract a set of stores when $n > 1$?

2.5.1 Independent Attribute Analyses

Suppose value sets are abstracted by $\alpha_{val} : \wp(\text{Value}) \to \text{A}$. The *independent attribute method* models a set of stores S at program point p by mapping each variable V_i to an abstraction of the set of values it takes in all the stores of S. This abstract value is thus independent of all other variables, hence the term "independent attribute". For example, $\{[X \mapsto 1, Y \mapsto 2], [X \mapsto 3, Y \mapsto 1]\})$ would be modelled by $[X \mapsto odd, Y \mapsto \top]$.

Formally, we model

$$S \in \wp(\text{Store}) = \wp(\{V_1,\ldots,V_n\} \to \text{Value})$$

by a function

$$\text{abs}_p \in \text{Abs} = \{V_1,\ldots,V_n\} \to A$$

The store abstraction function $\alpha_{sto} : \wp(\text{Store}) \to \text{Abs}$ is defined by

$$\alpha_{sto}(S) = [V_i \mapsto \alpha_{val}(\{s(V_i) \mid s \in S\})]_{i=1,\ldots,n}$$

For example, consider an even-odd analysis of a program with variables X, Y, Z. The independent attribute method would abstract a set of two stores as follows:

$$\begin{array}{c}\alpha_{sto}(\{[X \mapsto 1, Y \mapsto 2, Z \mapsto 1], [X \mapsto 2, Y \mapsto 2, Z \mapsto 1]\}) \quad = \\ [X \mapsto \alpha_{val}(\{1,2\}), Y \mapsto \alpha_{val}(\{2\}), Z \mapsto \alpha_{val}(\{1\})] \quad = \\ [X \mapsto \top, Y \mapsto even, Z \mapsto odd]\end{array}$$

The independent attribute method abstracts each variable independently of all others, and so allows "cross over" effects. An example:

$$\alpha(\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}) = [X \mapsto \top, Y \mapsto \top] = \alpha(\{[X \mapsto 1, Y \mapsto 2], [X \mapsto 2, Y \mapsto 1]\})$$

This loses information about relationships between X's and Y's values, e.g. whether or not they always have the same parity.

2.5.2 Relational Analyses

Relations and predicates Abstract value abs_p is an abstraction of the set of stores acc_p, so the question arises as to how to represent it by a lattice element. An approach used in [Cousot, 1977a], [Cousot, 1977b] is to describe acc_p and its approximations abs_p by predicate calculus formulas. For instance the set of two stores $\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}$ above could be approximately described by the formula:

$$(odd(X) \wedge odd(Y)) \vee (even(X) \wedge even(Y))$$

More generally, suppose Store = $\{\mathrm{V}_1,...,\mathrm{V}_n\} \rightarrow$ Value. Clearly Store is isomorphic to Value^n, the set of all n-tuples of values. Thus any set of stores i.e. any element of $\wp$(Store)) can be interpreted as a set of n-tuples. For example, store set $\{[X \mapsto 1, Y \mapsto 1], [X \mapsto 2, Y \mapsto 2]\}$ corresponds to {(1,1), (2,2)}. Thus a store set is essentially a set of n-tuples or, in other words, an n-ary *predicate* or *relation*.

For program point p, the accumulating semantics defines relation $\mathrm{acc}_p(\mathrm{v}_1,\ldots,\mathrm{v}_n)$ to be true just in the case that $(\mathrm{v}_1,...,\mathrm{v}_n)$ is a tuple of values which can occur at p in one or more computations on the given initial input data. This is the weakest possible relation among variables that always holds at point p.

Relational Analyses These use more sophisticated methods to approximate $\wp$(Store), which can give more precise information. Examples of practically motivated program analysis problems that require relational information include aliasing analysis in Pascal, the recognition of possible substructure sharing in Lisp or Prolog, and interference analysis.

For an example not naturally represented by independent attributes, suppose we wish to find out which of a program's variables always assume the same value at a given program point p. A suitable abstraction of a set of stores is a *partition* π_p that divides the program's variables into equivalence classes, so any one class of π_p contains all variables that have the same value at p. The effect of an assignment such as "p: X:=Y; **goto** q" is that π_q is obtained from π_p by removing X from its previous equivalence class and adding it to Y's class.

Intensional versus extensional descriptions Above we represented store set

$$\{[X \mapsto 1, Y \mapsto 1],\ [X \mapsto 2, Y \mapsto 2]\}$$

by the binary relation {(1,1), (2,2)}, and approximated it by the superset {(x,y) | x and y are both even or both odd}, denoted by the predicate calculus formula

$$(odd(X) \wedge odd(Y)) \vee (even(X) \wedge even(Y))$$

The view of "predicate as a set of tuples" and "predicate as a formula" is exactly the classical distinction between the *extensional* and the *intension*al views of a predicate.

Descriptions by predicate calculus formulas must of necessity be only approximate, since there are only countably many formulas but uncountably many sets of stores (if we assume an infinite variable value set). In terms of predicate calculus formulas, for each program point p the appropriate formulation of a safe approximation is that acc_p *logically implies* abs_p. In terms of sets of n-tuples: each acc_p is a subset of the set of all tuples satisfying abs_p.

2.5.3 Abstract Interpretation and Predicate Transformers

The new view of the accumulating semantics is: given a program and a predicate describing its input data, the accumulating semantics maps every program point to the smallest *relation among variables that holds whenever control reaches that point.*

From this viewpoint, the function $\mathrm{next}_{p,q} : \wp(\mathrm{Store}) \rightarrow \wp(\mathrm{Store})$ is clearly the *forward predicate transformer* [Dijkstra, 1976] associated with transitions from p to q. Further, acc_p is clearly *the strongest postcondition* holding at program point p over all computations on input data satisfying the program's input precondition.

Program verification amounts to proving that each acc_p logically implies a user-supplied program assertion for point p. Note however that this abstract interpretation framework says nothing at all about program termination. This approach is developed further in [Cousot, 1977b] and their subsequent works.

Backwards analyses All this can easily be dualised: the *backward predicate transformer* $\mathrm{next}_{p,q}^{-1} : \wp(\mathrm{Store}) \rightarrow \wp(\mathrm{Store})$ is just the inverse of $\mathrm{next}_{p,q}$, and given a *program postcondition* one may find the *weakest precondition* on program input sufficient to imply the postcondition at termination. For the simple imperative language, a backward accumulating semantics is straightforward to construct. For the example program

```
A: while $n \neq 1$ do
    B: if $n$ even
        then (C: $n := n \div 2$; D: )
        else (E: $n := 3 * n + 1$; F: )
      fi
    od
G:
```

the appropriate equations are:

$$\begin{aligned}
\mathrm{acc}_A &= (\{1\} \cap \mathrm{acc}_G) \cup (\{0,2,3,4,\ldots\} \cap \mathrm{acc}_B) \\
\mathrm{acc}_B &= (\mathrm{acc}_C \cap \mathit{Evens}) \cup (\mathrm{acc}_E \cap \mathrm{Odds}) \\
\mathrm{acc}_C &= \{n \mid n \div\ 2 \in\ \mathrm{acc}_D\} \\
\mathrm{acc}_D &= (\{1\} \cap \mathrm{acc}_G) \cup (\{0,2,3,4,\ldots\} \cap \mathrm{acc}_B) \\
\mathrm{acc}_E &= \{n \mid 3n+1 \in \mathrm{acc}_F\} \\
\mathrm{acc}_F &= (\{1\} \cap \mathrm{acc}_G) \cup (\{0,2,3,4,\ldots\} \cap \mathrm{acc}_B) \\
\mathrm{acc}_G &= S_{final}
\end{aligned}$$

where acc_p is the set of all stores at point p that cause control to reach point G with a final store in S_{final}.

Such a backward accumulating semantics can, for example, provide a basis for an analysis that detects the set of states that may lead to an error. More generally backward analyses (although not the one shown here) may provide a basis for "future sensitive" analysis such as *live variables*, where variable X is "semantically live" at point p if there is a computation sequence starting at p and later referencing X's value. This is approximated by: X is "syntactically live" if there is a program path from p to a use of X's value. Section 3 contains an example of live variable analysis for functional programs.

Many analysis problems can be solved by either a forwards or a backwards analysis. There can, however, be significant differences in efficiency.

Backwards analysis of functional programs The backwards accumulating semantics is straightforward for imperative programs, partly because of its close connections with the well studied weakest preconditions [Dijkstra, 1976], and because the state transition function is monadic. It is semantically less well understood, however, for functional programs, where recent works include [Hughes, 1987], [Dybjer, 1987], [Wadler, 1987], and [Nielson, 1989]. Natural connections between backwards analyses and continuation semantics are seen in [Nielson, 1982] and [Hughes, 1987].

2.6 Important Points from Earlier Sections

In the above we have employed a rather trivial programming language so as to motivate and illustrate one way to approximate real computations by computations over a domain of abstract values: Cousot's accumulating semantics. Before proceeding to abstract interpretation of more interesting languages we recapitulate what has been learned so far.

- Computations in the abstract world are at best *semihomomorphic* models of corresponding computations in the world of actual values.
- *Safety* of an abstract interpretation is analogous to *reliability* of a numerical analyst's results: the obtained results must always lie within specified confidence intervals (usually "one-sided intervals" in the case of program analysis).
- To obtain safe results for specific applications it is essential to understand the interpretation of the abstract values and their relation to actual computational values. One example is *modality*, e.g. "all computations" versus "some computations".
- Abstract values often do not contain enough information to determine the outcome of tests, so abstract interpretation must achieve the effect of simulating a *set* of real computations.
- Computations on *complete lattices* of abstract values appropriately model computations on real values.
- The partial order on these lattices expresses the degree of precision in an approximate description, and is *quite different* from the traditional Scott-style ordering based on filling in incomplete information.
- Termination can be achieved by choosing lattices without infinite ascending chains.
- *Best* approximations to real computations exist in principle, but may be uncomputable.
- There are close connections between the "accumulating semantics" and the predicates and predicate transformers (both forwards and backwards) used in program verification.

2.7 Towards Generalizing the Cousot Framework

Abstract interpretation is a semantics-based approach to program analysis, but so far we have only dealt with a single, rather trivial language.