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

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

and

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

with carriers D, E and operators a_i, b_i, a *homomorphism* is a function $\beta : D \to E$ 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

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

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

$$\begin{array}{lcl}
\text{AbState} & = & \text{Program point} \times \text{AbStore} \\
\text{AbStore} & = & \text{Variable} \rightarrow \{+,-,\ ?\} \\
\textbf{next} & : & \text{AbState} \rightarrow \text{AbState}
\end{array}$$

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

$$\text{p} : Y := X + Y;\ \textbf{goto}\ \text{q}$$

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

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

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:

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

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

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

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

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

to work on *sets of states*, namely

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

defined by

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

together with an abstraction function α: $\wp(\text{State}) \to \text{AbState}$. 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:

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

[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.

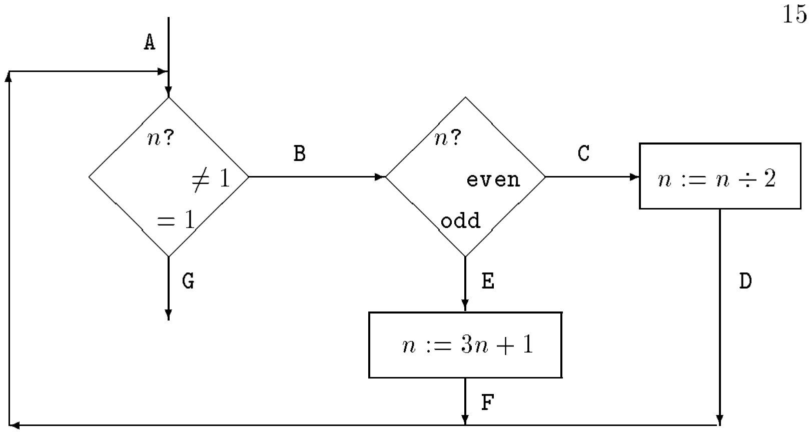
The idea of an "instrumented" semantics is introduced and correctness is discussed. This section is problem-oriented, with simulation techniques chosen ad hoc to fit the analysis problem and the language being analysed. It thus centers more around programs' operational behaviour than the structure of 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 $\div$ 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

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

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 $\alpha : \wp(Store) \to \text{Abs}$ 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 $\top, \perp$ for the least, resp. greatest elements of Abs.

An abstraction function is most often used together with a dual *concretization* function $\gamma : \text{Abs} \to \wp(\text{Store})$, 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

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

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 $\mathrm{acc}_p \subseteq \mathrm{Store}$.

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 $\wp(\mathrm{Store})$, and it is easy to see that their fixpoint is exactly the tuple of sets of reachable stores as defined above.

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

$$\text{acc}_G = (\text{acc}_A \cup \text{acc}_D \cup \text{acc}_F) \cap \{1\}$$

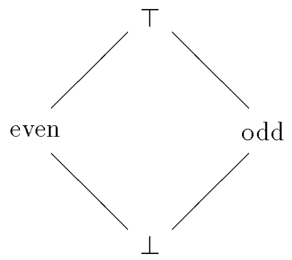
The equation set can be derived mechanically from the given program's syntax, e.g. as seen in [Cousot, 1977a] or [Nielson, 1982].

2.2.3 Abstract Interpretation of the Example Program

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

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

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.



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:

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

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:

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

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**

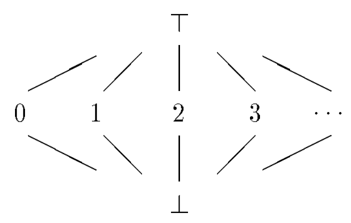
which avoids the two tests every time n is odd. In practice, one of the main reasons for doing abstract interpretation is to find out when such optimizing transformations may be performed.

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 $\mathrm{Abs} = \{\top, \perp, 0, 1, 2, \ldots\}$ 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 $\mathrm{Abs} = (N, \geq)$.

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):

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

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

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

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

□

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 op : $\mathbb{N} \to \mathbb{N}$, and extend it, by "pointwise lifting" to sets of numbers, yielding $\wp\text{op} : \wp(\mathbb{N}) \to \wp(\mathbb{N})$ where

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

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

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

or, diagrammatically:

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

By the conditions and lemma this is equivalent to

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

corresponding to diagram:

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

Intuitively, for any subset $N \subseteq \mathbb{N}$, applying the induced abstract operation $\mathbf{op}$ 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 $\mathbf{op}$ 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 $\alpha : \wp(\mathbb{N}) \rightarrow \text{Abs}$ and $\gamma : \text{Abs} \rightarrow \wp(\mathbb{N})$ satisfying the three conditions above, define the operator *induced* by op to be $\mathbf{op} : \text{Abs} \rightarrow \text{Abs}$ 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 $f : \text{Abs} \rightarrow \text{Abs}$ 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 $\text{next}_{p,q}\colon\wp(\text{Store}) \rightarrow \wp(\text{Store})$:

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

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

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

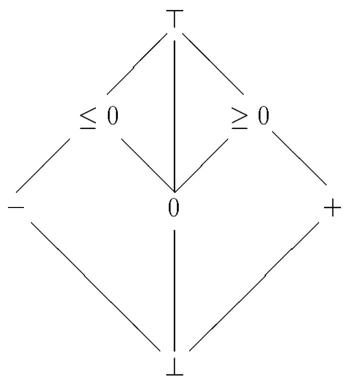
for all S.

2.3.4 An Example: the Rule of Signs

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

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

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

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

and abstraction function

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

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

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

2.3.5 Composition of Safety Diagrams

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

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

and

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

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

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

On the other hand the diagrams for the *induced* approximations to base functions *cannot* be so composed, since the best approximation to $\wp\mathrm{op}' \circ \wp\mathrm{op}$ may be better than the composition of the best approximations to $\wp\mathrm{op}$ and $\wp\mathrm{op}'$. (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

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

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 $\wp(\text{Store})$ 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 = $\{V_1,...,V_n\} \to$ 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 $\alpha_{val} : \wp(\text{Value}) \to$ 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 $\alpha_{sto} : \wp(\text{Store}) \to$ 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 = $\{\text{V}_1,...,\text{V}_n\} \rightarrow$ 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 $\text{acc}_p(\text{v}_1,\ldots,\text{v}_n)$ to be true just in the case that $(\text{v}_1,...,\text{v}_n)$ 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:

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

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.

Rather than redo the same work for every new language, we set the foundations for developing a general framework based on *denotational semantics*. This is a widely used and rather general formalism for defining the semantics of programming languages (see [Schmidt, 1986,Stoy, 1977, Gordon, 1979,Nielson, 1992a]). Other possibilities include axiomatic and structural operational semantics [Plotkin, 1981], and natural semantics [Kahn, 1987, Nielson, 1992a]. They are also general frameworks, but ones in which few applications to abstract interpretation have been developed (although operational semantics seems especially promising).

The approach will be developed stepwise. First, a denotational semantics is given for essentially the simple imperative language seen before. This is then factored into two stages, into a *core semantics* and an *interpretation*. The interpretation specifies the details relevant to a specific (standard or abstract) interpretation of the program's values and operations, and the core semantics specifies those parts of the semantics that are to be used in the same way for all interpretations. It is then shown how, given a fixed core semantics, interpretations may be partially ordered with respect to "degree of abstractness", and it is shown that a concrete interpretation's execution results are always compatible with those of more abstract interpretations. This provides a basis for formally proving the safety of an analysis, for example by showing that a given abstract interpretation is an abstraction of the accumulating interpretation. The last step is to describe briefly a way to generalize this approach to denotational definitions of other languages; this gives a bridge to the development of section 3.

Earlier papers using this approach include [Donzeau-Gouge, 1978], [Nielson, 1982], [Jones, 1986].

Denotational semantics has three basic principles:

1. Every syntactic phrase in a program has a *meaning*, or *denotation*.
2. Denotations are well-defined mathematical objects (often higher-order functions).
3. The meaning of a compound syntactic phrase is a mathematical combination of the meanings of its immediate subphrases.

The last assumption is often called *compositionality* or, according to Stoy, *the denotational assumption*. Phrase meanings are most often given by expressions in the typed lambda calculus, although other possibilities exist. A denotational language definition consists of the following parts:

- a collection of *domain definitions*, to be used as types for the lambda expressions used to define phrase meanings

- a collection of *semantic functions*, usually one for each syntactic phrase type
- a collection of *semantic rules* defining them, expressing the meanings of syntactic phrases in terms of the meanings of their substructures, usually by lambda expressions.

A tiny denotational language definition For an example, consider a language of while-programs with abstract syntax

$$
\begin{array}{llll}
\mathrm{c}: & \mathrm{Cmd} & ::= & \mathrm{x} := \mathrm{e} \mid \mathrm{c}\ ;\ \mathrm{c'} \\
 & & \mid & \textbf{if}\ \mathrm{e}\ \textbf{then}\ \mathrm{c}\ \textbf{else}\ \mathrm{c'} \mid \textbf{while}\ \mathrm{e}\ \textbf{do}\ \mathrm{c} \\
\mathrm{e}: & \mathrm{Exp} & ::= & \mathrm{x} \mid \mathrm{Constant} \mid \mathrm{op}(\mathrm{e}_1, \ldots, \mathrm{e}_n)
\end{array}
$$

where x is assumed to range over a set Var of variables, op is a basic operation (e.g. $+, -, *, \div$), and Constant denotes any member of a not further specified set Value of values. The part of a denotational semantics relevant to commands could be as follows, where the traditional "semantic parentheses" $[\![$ and $]\!]$ enclose syntactic arguments. In the following Store and Value are as before except that for concreteness we specify Value is the set of numbers. Each can be thought of as a "flat" Scott domain with $d \sqsubseteq d'$ iff $d = \bot$ or $d = d'$.

Specifying the semantics of a **while** loop requires (as usual) evaluating a fixpoint over domain Store $\rightarrow$ Store, ordered "pointwise": s $\sqsubseteq$ s$'$ iff s(x) $\sqsubseteq$ s$'$(x) for all variables x.

Domain definitions

$$
\begin{array}{l}
\mathrm{s}: \mathrm{Store} = \mathrm{Var} \rightarrow \mathrm{Value} \\
\quad\ \ \mathrm{Value} = \mathrm{Number}
\end{array}
$$

Types of semantic and auxiliary functions

$$
\begin{array}{l}
\mathrm{C} : \mathrm{Cmd} \rightarrow \mathrm{Store} \rightarrow \mathrm{Store} \\
\mathrm{E} : \mathrm{Exp} \ \ \rightarrow \mathrm{Store} \rightarrow \mathrm{Value}
\end{array}
$$

Semantic rules

$$
\begin{array}{lcl}
\mathrm{C}[\![\mathrm{x} := \mathrm{e}]\!] & = & \lambda \mathrm{s}\ .\ \mathrm{s}[\mathrm{x} \mapsto \mathrm{E}[\![\mathrm{e}]\!]\ \mathrm{s}] \\
\mathrm{C}[\![\mathrm{c}\ ;\ \mathrm{c'}]\!] & = & \lambda \mathrm{s}\ .\ \mathrm{C}[\![\mathrm{c'}]\!](\mathrm{C}[\![\mathrm{c}]\!]\ \mathrm{s}) \\
\mathrm{C}[\![\textbf{if}\ \mathrm{e}\ \textbf{then}\ \mathrm{c}\ \textbf{else}\ \mathrm{c'}]\!] & = & \lambda \mathrm{s}\ .\ \mathrm{E}[\![\mathrm{e}]\!]\ \mathrm{s} \neq 0 \rightarrow \mathrm{C}[\![\mathrm{c}]\!]\ \mathrm{s},\ \mathrm{C}[\![\mathrm{c'}]\!]\ \mathrm{s} \\
\mathrm{C}[\![\textbf{while}\ \mathrm{e}\ \textbf{do}\ \mathrm{c}]\!] & = & \mathrm{fix}\ \lambda\phi\ .\ \lambda \mathrm{s}\ .\ \mathrm{E}[\![\mathrm{e}]\!]\ \sigma \neq 0 \rightarrow \phi(\mathrm{C}[\![\mathrm{c}]\!]\ \mathrm{s}),\ \mathrm{s}
\end{array}
$$

Note that all the rules are compositional.

2.7.1 Factoring a Denotational Semantics

The principle of compositionality provides an ideal basis for generalizing the denotational semantics framework to allow alternate, nonstandard interpretations in addition to the "standard" semantics defining the meanings of programs. The idea is to decompose a denotational language definition into two parts:

- a *core semantics*, containing semantic rules and their types, but using some uninterpreted domain names and function symbols, and
- an *interpretation*, filling out the missing definitions of domain names and function symbols

The interpretation is clearly a many-sorted algebra. Examples include the "standard interpretation" defining normal program execution, an "accumulating interpretation" analogous to the accumulating semantics of section 2, and as well more approximate and effectively computable interpretations suitable for program analysis, e.g. for parity analysis.

Scott Domains versus Complete Lattices In denotational semantics the denoted values are nearly always elements of "domains" in the sense of Dana Scott and others. These are cpo's (complete partial orders with $\perp$), usually required to be algebraic. For material on domains see the list of references in the beginning of this chapter.

On the other hand for abstract interpretation purposes, it is usual to use complete lattices for reasons mentioned earlier. There is no basic conflict here since cpo's include complete lattices as a special case. On the other hand, the interpretation of the partial order is somewhat different in semantics than in abstract interpretation (as mentioned in section 2.4), so some care must be taken. This matter is further addressed in section 3.

An example factorized semantics For the imperative language above we obtain the following, where domains Sto and Val, and functions assign, seq, cond, while are unspecified:

Domain definitions

M_{Cmd}	$= \mathrm{Sto} \rightarrow \mathrm{Sto}$
M_{Exp}	$= \mathrm{Sto} \rightarrow \mathrm{Val}$
Sto, Val:	unspecified

Types of semantic and auxiliary functions

C	$: \mathrm{Cmd} \rightarrow \mathrm{M}_{Cmd}$
E	$: \mathrm{Exp} \rightarrow \mathrm{M}_{Exp}$

$$\begin{array}{ll}
\text{assign} & : \text{Var} \times \text{M}_{Exp} \rightarrow \text{M}_{Cmd} \\
\text{seq} & : \text{M}_{Cmd} \times \text{M}_{Cmd} \rightarrow \text{M}_{Cmd} \\
\text{cond} & : \text{M}_{Exp} \times \text{M}_{Cmd} \times \text{M}_{Cmd} \rightarrow \text{M}_{Cmd} \\
\text{while} & : \text{M}_{Exp} \times \text{M}_{Cmd} \rightarrow \text{M}_{Cmd}
\end{array}$$

Semantic rules

$$\begin{array}{ll}
\text{C}[\![\text{x := e }]\!] & = \text{assign(x, E}[\![\text{e}]\!]) \\
\text{C}[\![\text{c ; c' }]\!] & = \text{seq(C}[\![\text{c}]\!], \text{C}[\![\text{c'}]\!]) \\
\text{C}[\![\textbf{if}\text{ e }\textbf{then}\text{ c }\textbf{else}\text{ c'}]\!] & = \text{cond(E}[\![\text{e}]\!], \text{C}[\![\text{c}]\!], \text{C}[\![\text{c'}]\!]) \\
\text{C}[\![\textbf{while}\text{ e }\textbf{do}\text{ c}]\!] & = \text{while(E}[\![\text{e}]\!], \text{C}[\![\text{c}]\!])
\end{array}$$

The standard interpretation This is $\mathbf{I}_{std}$ = (Val, Sto; assign, seq, cond, while), defined by

Domains

Val = Number (the flat cpo)
Sto = Var → Val

Function definitions

$$\begin{array}{ll}
\text{assign} & = \lambda(\text{x, m}_e)\ .\ \lambda\text{s . s[x} \mapsto \text{m}_e\text{s]} \\
\text{seq} & = \lambda(\text{m}_{1c}, \text{m}_{2c})\ .\ \text{m}_{2c} \circ \text{m}_{1c} \\
\text{cond} & = \lambda(\text{m}_e, \text{m}_{1c}, \text{m}_{2c})\ .\ \lambda\text{s . m}_e\text{ s} \neq 0 \rightarrow \text{m}_{1c}\text{ s, m}_{2c}\text{ s} \\
\text{while} & = \lambda(\text{m}_e, \text{m}_c)\ .\ \text{fix } \lambda\phi\ .\ \lambda\text{s . m}_e\text{ s} \neq 0 \rightarrow \phi(\text{m}_c\text{ s), s}
\end{array}$$

2.7.2 The Even-odd Interpretation

With the current machinery a general formulation of the even-odd analysis of section 2.2.3 may be given as our first nonstandard interpretation:

$$\mathbf{I}_{parity} = \text{(Val, Sto; assign, seq, cond, while)}$$

where Val, etc. are given by:

Domains

Val = $\{\bot, \text{even}, \text{odd}, \top\}$ with partial order
$\bot \sqsubseteq \text{even} \sqsubseteq \top$ and $\bot \sqsubseteq \text{odd} \sqsubseteq \top$
Sto = Var → Val

Function definitions

$$\begin{array}{ll}
\text{assign} & = \lambda(\text{x, m}_e)\ \lambda\text{s . s[x} \mapsto \text{m}_e\text{s]} \\
\text{seq} & = \lambda(\text{m}_{1c}, \text{m}_{2c})\ .\ \text{m}_{2c} \circ \text{m}_{1c} \\
\text{cond} & = \lambda(\text{m}_e, \text{m}_{1c}, \text{m}_{2c})\ \lambda\text{s . m}_{1c}\text{ s} \sqcup \text{m}_{2c}\text{ s} \\
\text{while} & = \lambda(\text{m}_e, \text{m}_c)\ .\ \text{fix } \lambda\phi\ \lambda\text{s . } \phi(\text{m}_c\text{s}) \sqcup \text{s}
\end{array}$$

Remarks

1. This is an *independent attribute* approximation: a set of stores is modelled by mapping its variables' values independently to elements of Val.

2. For the conditional, no attempt is made to simulate the test. Instead, the best description fitting both the then and else branches is produced.

3. The fixpoint clearly models the one in the standard interpretation, again without simulating the test. Termination is assured since Sto has finite height and any one program has a finite number of variables.

2.7.3 The Accumulating Semantics as an Interpretation

The accumulating semantics seen earlier was only given by example. With the current machinery a general formulation may be given: $\mathbf{I}_{acc}$ = (Val, Sto; assign, seq, cond, while) where Val, etc. are given by:

Domains

Val	=	$\wp$(Number)	
Sto	=	Storeset	typical element S
Storeset	=	$\wp$(Var $\rightarrow$ Val)	a set of stores

Function definitions

$$
\begin{array}{lcl}
\text{assign} & = & \lambda(\text{x}, \text{m}_e)\ .\ \lambda\text{S}\ .\ \{\text{s}[\text{x} \mapsto \text{v}] \ \mid \text{s} \in \text{S and v} \in \text{m}_e\{\text{s}\}\} \\
\text{seq} & = & \lambda(\text{m}_{1c}, \text{m}_{2c})\ .\ \text{m}_{2c} \circ \text{m}_{1c} \\
\text{cond} & = & \lambda(\text{m}_e, \text{m}_{1c}, \text{m}_{2c})\ .\ \lambda\text{S}\ . \\
& & \text{m}_{1c}\ (\{\text{s} \in \text{S} \mid\ 0 \notin \text{m}_e\{\text{s}\}\}) \cup \text{m}_{2c}\ (\{\text{s} \in \text{S} \mid 0 \in \text{m}_e\{\text{s}\}\}) \\
\text{while} & = & \lambda(\text{m}_e, \text{m}_c)\ .\ \text{fix}\ \lambda\phi .\ \lambda\text{S}\ . \\
& & \{\text{s} \in\text{S} \mid\ \text{m}_e\ \text{s} = 0\} \cup \phi(\ \text{m}_c\ \{\text{s} \in \text{S} \mid \text{m}_e\ \text{s} \neq 0\})
\end{array}
$$

Here the denotation of C[[Cmd]] has been "lifted" from Sto $\rightarrow$ Sto to $\wp$(Sto) $\rightarrow$ $\wp$(Sto), so it now transforms a set of current states into the set of possible next states. To relate this to the earlier accumulating semantics of section 2.2, consider for example the program fragment "C : n := n÷2; D" of section 2.2.2. Then

$$\text{C}[\![\text{n} := \text{n}\div 2]\!]\text{acc}_C = \text{acc}_D$$

In general, C[[c]] realizes the same transformation on store sets as defined by the data flow equations for command c.

2.8 Proving Safety by Logical Relations

2.8.1 Safety from a Denotational Viewpoint

Suppose we are given two interpretations

I = (Val, Sto; assign, seq, cond, while)

and

I' = (Val', Sto', assign', seq', cond',while')

and a pair of abstraction functions

$\beta = (\beta_{val} : \text{Val} \rightarrow \text{Val'}, \beta_{sto} : \text{Sto} \rightarrow \text{Sto'})$

where β_{val} and β_{sto} are monotone. We write this as β: **I** $\rightarrow$ **I'**. Define s $\sqsubseteq$ s' to hold iff s (X) $\sqsubseteq$ s'(X) for all variables X $\in$ Var.

Definition $\beta : \mathbf{I} \rightarrow \mathbf{I'}$ is *safe* if for all c $\in$ Cmd and s $\in$ Sto

$\beta_{sto}(\text{C}_I[\![\text{c}]\!]\ \text{s}) \sqsubseteq \text{C}_{I'}[\![\text{c}]\!]\ (\beta_{sto}\ \text{s})$

where C_I : Sto $\rightarrow$ Sto is the semantic function obtained using assign, seq, etc. and $\text{C}_{\mathbf{I}'}$ is analogous but using assign', seq', etc.

Recall that $a \sqsubseteq a'$ signifies that a' is a more approximate description than a. This definition says that the result of computing in the "real world" and then abstracting the resulting store gives a result that is safely approximable by first abstracting the real world's initial store, and then computing entirely in the "abstract world". It is thus simply a reformulation in denotational terms of the earlier condition on safe approximation of transitions:

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

where p and q are (resp.) the entry and exit points of command c (and α plays the role of β).

2.8.2 A Sufficient Local Condition for Safety

While pleasingly general, this definition is unfortunately global: it quantifies over all commands c. It would be strongly desirable to have *local* conditions on assign, seq, etc. sufficient to guarantee safety in the sense above. This can be done, but requires first setting up a bit of descriptive machinery. The problem is that denotational definitions use higher order functions and cartesian products, whereas the earlier conditions for safety were developed only for first order domains.

The solution we present is an instance of *logical relations*, an approach to relating values in different but simlarly structured domains that will be developed further in Section 3.3.

Suppose we have two interpretations **I** and **I**' of our simple imperative core semantics, related by $\beta = (\beta_{val} : \text{Val} \rightarrow \text{Val}', \beta_{sto} : \text{Sto} \rightarrow \text{Sto}')$ where β_{val} and β_{sto} are again monotone. Our goal is to see how to extend β to apply to all domains built up from those of **I** and **I**'. Suppose further that domain A is built by $\times$ and $\rightarrow$ from the domains of **I**, and a corresponding domain A' is built in the same way from the domains of **I'**. We define the binary relation a $\leq_\beta a'$, which will hold whenever a' $\in$ A' is a safe approximation of the corresponding element a $\in$ A.

Definition Suppose $\beta = \beta_{val} : \mathrm{Val} \rightarrow \mathrm{Val}'$ and $v \in \mathrm{Val}, v' \in \mathrm{Val}'$. Then

1. $v \leq_\beta v'$ if and only if $\beta_{val}(v) \sqsubseteq v'$ and similarly for $\beta = \beta_{sto} : \mathrm{Sto} \rightarrow \mathrm{Sto}'$.
2. Let $(a, b) \in$ A$\times$B, $(a', b') \in$ A'$\times$B' and β_A : A $\rightarrow$ A', β_B : B $\rightarrow$ B' be monotone. Then
$$(a, b) \leq_\beta (a', b')$$
if and only if
$$a \leq_{\beta_A} a' \text{ and } b \leq_{\beta_B} b'.$$
3. Let f : A $\rightarrow$ B, g' : A' $\rightarrow$ B' and β_A : A $\rightarrow$ A', β_B : B $\rightarrow$ B' be monotone. Then
$$f \leq_\beta g \text{ if and only if}$$
$$\forall a \in A\, \forall a' \in A' (a \leq_{\beta_A} a' \text{implies} fa \leq_{\beta_B} ga').$$

□

This notation allows an alternate characterization of safety.

Lemma $\beta : \mathbf{I} \rightarrow \mathbf{I'}$ is safe if and only if $\mathrm{C}_{\mathbf{I}}[\![\mathrm{c}]\!] \leq_\beta \mathrm{C}_{\mathbf{I}'}[\![\mathrm{c}]\!]$ for all c $\in$ Cmd

Proof "If": by 3, $\mathrm{C}_{\mathbf{I}}[\![\mathrm{c}]\!] \leq_\beta \mathrm{C}_{\mathbf{I}'}[\![\mathrm{c}]\!]$ holds if and only if $\mathrm{C}_{\mathbf{I}}[\![\mathrm{c}]\!]\mathrm{s} \leq_{\beta_{sto}} \mathrm{C}_{\mathbf{I}'}[\![\mathrm{c}]\!]$ s' whenever s $\leq_{\beta_{sto}}$ s'. In particular we have s $\leq_\beta \beta_{sto}(\mathrm{s})$, hence

$$\beta_{sto}(\mathrm{C}_I[\![\mathrm{c}]\!]\mathrm{s}) \sqsubseteq \mathrm{C}_{I'}[\![\mathrm{c}]\!]\beta_{sto}(\mathrm{s})$$

so β is safe (as defined before).

"Only if": if β is safe and s $\leq_\beta$ s' then

$$\beta_{sto}(\mathrm{C}_I[\![\mathrm{c}]\!]\ \ \mathrm{s})\ \sqsubseteq \mathrm{C}_{I'}[\![\mathrm{c}]\!]\ (\beta_{sto}\mathrm{s})\ \sqsubseteq \mathrm{C}_{I'}[\![\mathrm{c}]\!]\ \mathrm{s'}$$

by monotonicity of $\mathrm{C}_{I'}[\![\mathrm{c}]\!]$ (easily verified). □

It is now natural to extend the definition of $\leq_\beta$ to allow comparison of interpretations. Given this, we are finally ready to define a local safety condition which implies global safety. Proof is by a straightforward induction on program syntax.

Definition Let $\beta : \mathbf{I} \rightarrow \mathbf{I'}$ be defined as above. Then $\mathbf{I} \leq_\beta \mathbf{I'}$ if and only if

$$\begin{aligned} assign &\leq_\beta assign', \\ seq &\leq_\beta seq', \\ cond &\leq_\beta cond' \\ while &\leq_\beta while'. \end{aligned}$$

Theorem $\beta : \mathbf{I} \rightarrow \mathbf{I'}$ is safe if $\mathbf{I} \leq_\beta \mathbf{I'}$.

A straightforward generalization of these ideas to arbitrary denotational definitions provides a very general framework for program analysis by interpreting programs over nonstandard domains, and gives a way to show that one abstract interpretation is a refinement of and compatible with another. This observation is the starting point for the development in the section below.

For a very simple example, the following is easy to show. Its significance is that the accumulating semantics is a faithful extension of with the standard semantics.

Lemma Let $\mathbf{I}_{std}$ = (Val, Sto; assign, seq, cond, while) and $\mathbf{I}_{acc}$ = (Val', Sto'; assign', seq', cond', while') be the standard and accumulating semantics. Then $\mathbf{I}_{std} \leq_\beta \mathbf{I}_{acc}$, where β_{val}(v) = {v} for v $\in$ Val and β_{sto}(s)= {s} for s $\in$ Sto.

3 Abstract Interpretation Using a Two-Level Metalanguage

In the previous section we have given a survey of many concepts in abstract interpretation. We have stressed that abstract interpretation should be *generally applicable* to programs in a wide class of languages, that it should always produce *correct* properties and that it should always *terminate*.

To ensure the *general applicability* of abstract interpretation we adopt the framework of denotational semantics. Most modern approaches to denotational semantics stress the role of a formal metalanguage in which the semantics is defined. So rather than regarding denotational semantics as directly mapping programs to mathematical domains one regards denotational semantics as factored through the metalanguage. This is illustrated by the upper half of Figure 1: First one uses the semantic equations to expand programs into terms in the metalanguage and then one interprets the terms in the metalanguage as elements in the mathematical domains used in denotational semantics.

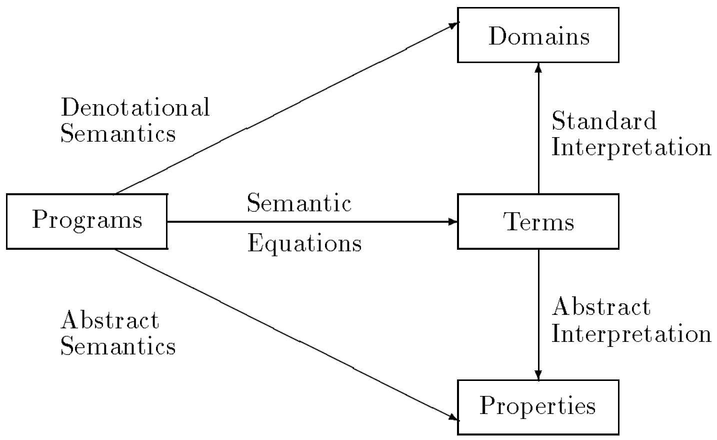


Fig. 1. The Role of the Metalanguage

Following our informal presentation of a parameterised semantics in Subsection 2.7 we take a similar factored approach to abstract interpretation. This is illustrated in the lower half of Figure 1 which furthermore stresses that the semantic equations are the same. The Semantic Equations thus correspond to the core semantics of Subsection 2.7. From the point of view of developing a general theory, the focus will be on the metalanguage but we trust that the reader will be able to see for himself that the development is of wider applicability than just the simple metalanguage considered here. Its syntax is defined in Subsection 3.1 and we give example interpretations (i.e. semantics) in Subsection 3.2.

Correctness of abstract interpretation will be our guide throughout the development. In Subsection 3.3 we therefore give a structural definition of correctness relations between interpretations thereby extending the development surveyed in Subsection 2.8. As an example we define correctness relations between the standard interpretation and one of the abstract interpretations defined in Subsection 3.2.

Closely related to the question of correctness is the question of whether best *induced* property transformers exist over the abstract domains. We treat this in Subsection 3.4 and we consider the easier case of relating abstract interpretations to one another as well as the harder case of relating an abstract interpretation to the standard interpretation. Induced property transformers need not terminate but are none the less useful as guides in determining the degree of approximation that will be needed to ensure termination.

The *termination* aspect motivates the study in Subsection 3.5 of coarser versions of the induced property transformers which have the advantage of leading to analyses that will always terminate. Subsection 3.6 concludes by mentioning some generalisations that are possible [Nielson, 1989] and by discussing some issues that have not yet been incorporated in this treatement.

3.1 Syntax of Metalanguage

Most metalanguages for denotational semantics are based on some version of the λ-calculus. Depending on the kind of mathematical foundations used for denotational semantics the metalanguage may be without explicit types or it may have explicit types. We shall not pay great attention to this difference and in many instances the various algorithms for polymorphic type inference may be used to introduce types into an untyped notation. As our starting point we thus assume that our metalanguage is a small typed λ-calculus.

Definition 3.1.1. The Typed λ-Calculus has types $t{\in}T$ and expressions $e{\in}E$ given by

$$t ::= \mathtt{A}_{\mathrm{i}} \mid t{\times}t \mid t{\rightarrow}t$$

$$\begin{aligned} e ::= {} & \mathtt{f}_{\mathrm{i}}[t] \mid \langle e,e\rangle \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid \\ & \lambda\mathtt{x}_{\mathrm{i}}[t].e \mid e(e) \mid \mathtt{x}_{\mathrm{i}} \mid \mathtt{fix}\ e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \end{aligned}$$

Concerning types we have base types $\mathtt{A}_{\mathrm{i}}$ where i ranges over a countable index set (say I) and we have product and function space. Here $\times$ binds more tightly than $\rightarrow$ and both associate to the right. We shall not specify the details of the countable index set but we shall assume that we have booleans $\mathtt{A}_{\mathrm{bool}}$ (also written $\mathtt{Bool}$), integers $\mathtt{A}_{\mathrm{int}}$ (also written $\mathtt{Int}$) and other useful base types. However, nothing precludes us from having a base type $\mathtt{A}_{\mathrm{sto}}$ of machine stores and if the store contains just two values, say an integer and a boolean, we may write $\mathtt{A}_{\mathrm{int}\times\mathrm{bool}}$ for $\mathtt{A}_{\mathrm{sto}}$. (The difference between types like $\mathtt{A}_{\mathrm{int}\times\mathrm{bool}}$ and $\mathtt{A}_{\mathrm{int}}{\times}\mathtt{A}_{\mathrm{bool}}$ will become clear in Subsection 3.2.)

Concerning expressions we have basic expressions $\mathtt{f}_{\mathrm{i}}[t]$ of type t where again i ranges over a countable index set. (This index set need not be the same as the one used for the $\mathtt{A}_{\mathrm{i}}$ above but whenever we need to name it we shall use the same symbol I as above.) Again we expect to have familiar basic expressions like the truth values $\mathtt{f}_{\mathrm{true}}[\mathtt{Bool}]$ and $\mathtt{f}_{\mathrm{false}}[\mathtt{Bool}]$ (also written true[$\mathtt{Bool}$] and false[$\mathtt{Bool}$] or just true and false), integers like

$\mathtt{f}_0[\mathtt{Int}]$ (also written $\mathtt{0}[\mathtt{Int}]$ or just $\mathtt{0}$) and simple operations like equality $\mathtt{f}_=[\mathtt{Int}\times\mathtt{Int}\to\mathtt{Bool}]$ (also written $=[\mathtt{Int}\times\mathtt{Int}\to\mathtt{Bool}]$ or just $=$). Much as for the $\mathtt{A}_\mathrm{i}$ nothing prevents us from writing e.g. $\mathtt{f}_{\lambda\mathrm{x}.\lambda\mathrm{y}.\mathrm{x}=\mathrm{y}+\mathrm{y}}[\mathtt{Int}\to\mathtt{Int}\to\mathtt{Bool}]$ in order to clarify the intended meaning of some basic expression. The remaining constructs for expressions are pairing, selection of components, λ-abstraction, application, variables, fixed points and conditional. We shall assume that application binds more tightly than $\mathtt{fst}$, $\mathtt{snd}$ and $\mathtt{fix}$.

3.1.1 The use of underlining

We shall postpone the discussion about well-formedness of expressions in the typed λ-calculus because the syntax is not yet in a form that will suit our purpose: to prescribe a systematic approach to separating a denotational semantics into its core part and its interpretation part (to use the terminology of Subsection 2.7). To motivate this we recall from Section 2 that for a given programming language or example semantics there are some constructs that we might wish to interpret in different ways in different analyses whereas there are other constructs that we might as well interpret in the same way in all analyses. To indicate this distinction in a precise way we shall use the convention that *underlined* constructs are those that should have the freedom to be interpreted freely.

Beginning with the types we might consider a syntax as given by

$$t ::= \mathtt{A}_\mathrm{i} \mid \underline{\mathtt{A}}_\mathrm{i} \mid t\times t \mid t\to t$$

so that we would use $\underline{\mathtt{A}}_\mathrm{int}$ (also written $\underline{\mathtt{Int}}$) instead of $\mathtt{A}_\mathrm{int}$ whenever the integers are used in a context where we would like to perform abstract interpretation upon their values. Thus if we want to consider the store of an imperative programming language as a base type we will always use $\underline{\mathtt{A}}_\mathrm{sto}$ rather than $\mathtt{A}_\mathrm{sto}$. If we want to consider a structured version of the store where we have a fixed set $\mathtt{A}_\mathrm{ide}$ of identifiers and a fixed set $\mathtt{A}_\mathrm{val}$ of values we shall use $\mathtt{A}_\mathrm{ide}\to\underline{\mathtt{A}}_\mathrm{val}$ rather than e.g. $\mathtt{A}_\mathrm{ide}\to\mathtt{A}_\mathrm{val}$ or $\underline{\mathtt{A}}_\mathrm{ide}\to\underline{\mathtt{A}}_\mathrm{val}$.

However, this notation for types does not allow us to illustrate all the points we will need for a general theory of abstract interpretation although it would suffice for formalizing the development in Subsection 2.7. Examples include the discussion of forward versus backward analyses and the discussion of independent attribute versus relational methods. To cater for this we propose the syntax

$$t ::= \mathtt{A}_\mathrm{i} \mid \underline{\mathtt{A}}_\mathrm{i} \mid t\times t \mid t\,\underline{\times}\,t \mid t\to t \mid t\,\underline{\to}\,t$$

and we shall use the phrase *two-level types* for these. This will turn out to be a bit too liberal for our abilities so we shall need to impose various well-formedness conditions upon the types but in order to motivate them it is best to postpone this until they are needed for the technical development. In actual applications there might well be the need for distinguishing between various occurrences of $\underline{\times}$ and $\underline{\rightarrow}$ and one might then allow a notation like $\underline{\times}_{\mathrm{i}}$ and $\underline{\rightarrow}_{\mathrm{i}}$ where i ranges over some index set. However, as the theory hardly changes we shall leave this extension to the reader.

Turning to the expressions, a simple solution would be to keep the syntax of expressions as given in Definition 3.1.1 with the understanding that the types t in the basic expressions $\mathtt{f}_{\mathrm{i}}[t]$ now range over the larger set of two-level types. However, this is not quite in the spirit of the typed λ-calculus as we now have types without corresponding constructors and destructors. We shall therefore adopt a more comprehensive syntax of expressions by extending the use of underlining to the expressions.

Definition 3.1.2. The Two-level λ-Calculus has types $t \in T$ and expressions $e \in E$ given by

$$t ::= \mathtt{A}_{\mathrm{i}} \mid \underline{\mathtt{A}}_{\mathrm{i}} \mid t \times t \mid t \underline{\times} t \mid t \rightarrow t \mid t \underline{\rightarrow} t$$

$$\begin{aligned} e ::= {} & \mathtt{f}_{\mathrm{i}}[t] \mid \langle e,e \rangle \mid \underline{\langle} e,e \underline{\rangle} \mid \mathtt{fst}\ e \mid \underline{\mathtt{fst}}\ e \mid \mathtt{snd}\ e \mid \underline{\mathtt{snd}}\ e \mid \\ & \lambda \mathtt{x}_{\mathrm{i}}[t].e \mid \underline{\lambda} \mathtt{x}_{\mathrm{i}}[t].e \mid e(e) \mid e\underline{(}e\underline{)} \mid \mathtt{x}_{\mathrm{i}} \mid \\ & \mathtt{fix}\ e \mid \underline{\mathtt{fix}}\ e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \underline{\mathtt{if}}\ e\ \underline{\mathtt{then}}\ e\ \underline{\mathtt{else}}\ e \end{aligned}$$

Here the intention is that if e.g. e_1 is of type t_1 and e_2 is of type t_2 then $\langle e_1,e_2 \rangle$ will be of type $t_1 \times t_2$ and $\underline{\langle} e_1,e_2 \underline{\rangle}$ will be of type $t_1 \underline{\times} t_2$ and similarly for the other operators. We do not have two versions of $\mathtt{f}_{\mathrm{i}}[t]$ as $\mathtt{f}_{\mathrm{i}}[t]$ simply is a basic expression of the type indicated, nor do we have two versions of $\mathtt{x}_{\mathrm{i}}$ as $\mathtt{x}_{\mathrm{i}}$ simply is a placeholder for a 'pointer' to the enclosing $\lambda \mathtt{x}_{\mathrm{i}}$ or $\underline{\lambda} \mathtt{x}_{\mathrm{i}}$. In this notation an operation $\mathtt{seq}$ for sequencing two commands operating on a store $\underline{\mathtt{Sto}}$ might be defined by

$$\mathtt{seq} = \lambda \mathtt{x}_1[\underline{\mathtt{Sto} \rightarrow \mathtt{Sto}}].\ \lambda \mathtt{x}_2[\underline{\mathtt{Sto} \rightarrow \mathtt{Sto}}].\ \underline{\lambda} \mathtt{x}_{\mathrm{sto}}[\underline{\mathtt{Sto}}].\ \mathtt{x}_2\underline{(}\mathtt{x}_1\underline{(}\mathtt{x}_{\mathrm{sto}}\underline{))}$$

It will have type

$$(\underline{\mathtt{Sto} \rightarrow \mathtt{Sto}}) \rightarrow (\underline{\mathtt{Sto} \rightarrow \mathtt{Sto}}) \rightarrow (\underline{\mathtt{Sto} \rightarrow \mathtt{Sto}})$$

and may be used as in $\mathcal{C}[\![\mathrm{c}_1;\mathrm{c}_2]\!] = \mathtt{seq}(\mathcal{C}[\![\mathrm{c}_1]\!])(\mathcal{C}[\![\mathrm{c}_2]\!])$.

3.1.2 Combinators

The motivation behind the use of underlining was to separate the more 'dynamic' constructs that need to be interpreted freely from the more 'static' constructs whose interpretation never changes. Unfortunately the two-level λ-calculus is not in a form that makes this sufficiently easy. The problem is the occurence of free variables and especially those bound by $\underline{\lambda}$. This is not a novel problem and solutions have been found:

- When interpreting the typed λ-calculus in arbitrary cartesian closed categories one studies certain combinators ('categorical combinators') whose interpretation in a cartesian closed category is rather straightforward.
- When implementing functional languages one often transforms programs to combinator form before performing graph reduction.

This motivates:

Definition 3.1.3. The Two-level Metalanguage has types $t{\in}T$ and expressions $e{\in}E$ given by

$$t ::= \mathtt{A}_{\mathrm{i}} \mid \underline{\mathtt{A}_{\mathrm{i}}} \mid t{\times}t \mid t\underline{\times}t \mid t{\rightarrow}t \mid t\underline{\rightarrow}t$$

$$\begin{aligned} e ::= {} & \mathtt{f}_{\mathrm{i}}[t] \mid \langle e,e\rangle \mid \mathtt{Tuple}\langle e,e\rangle \mid \mathtt{fst}\ e \mid \mathtt{Fst}\ e \mid \mathtt{snd}\ e \mid \mathtt{Snd}\ e \\ & \mid \lambda\mathtt{x}_{\mathrm{i}}[t].e \mid \mathtt{Curry}\ e \mid e(e) \mid \mathtt{Apply}\langle e,e\rangle \mid \mathtt{x}_{\mathrm{i}} \mid \mathtt{Id}[t] \mid e \,\Box e \\ & \mid \mathtt{Const}[t]\ e \mid \mathtt{fix}\ e \mid \mathtt{Fix}\ e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{If}\langle e,e,e\rangle \end{aligned}$$

Here we have retained those expression constructs that were not underlined, we have replaced the underlined expression constructs by combinators, and we have added the new combinators $\mathtt{Id}[t]$, $\Box$ and $\mathtt{Const}[t]$. We shall regard application as binding more tightly than the prefixed operators ($\mathtt{fst}$, $\mathtt{Fst}$, $\mathtt{snd}$, $\mathtt{Snd}$, $\mathtt{Curry}$, $\mathtt{Const}[t]$, $\mathtt{fix}$ and $\mathtt{Fix}$). The intention with the combinators may be clarified by:

$$\begin{aligned} & \mathtt{Tuple}\langle e_1,e_2\rangle \equiv \underline{\lambda}\mathtt{x}_1.\underline{\langle} e_1\underline{(}\mathtt{x}_1\underline{)},\ e_2\underline{(}\mathtt{x}_1\underline{)}\underline{\rangle} \\ & \mathtt{Fst}\ e \equiv \underline{\lambda}\mathtt{x}_1.\underline{\mathtt{fst}}\ e\underline{(}\mathtt{x}_1\underline{)} \\ & \mathtt{Snd}\ e \equiv \underline{\lambda}\mathtt{x}_1.\underline{\mathtt{snd}}\ e\underline{(}\mathtt{x}_1\underline{)} \\ & \mathtt{Curry}\ e \equiv \underline{\lambda}\mathtt{x}_1.\underline{\lambda}\mathtt{x}_2.e\underline{(}\underline{\langle}\mathtt{x}_1,\mathtt{x}_2\underline{\rangle}\underline{)} \\ & \mathtt{Apply}\langle e_1,e_2\rangle \equiv \underline{\lambda}\mathtt{x}_1.e_1\underline{(}\mathtt{x}_1\underline{)}\underline{(}e_2\underline{(}\mathtt{x}_1\underline{)}\underline{)} \\ & \mathtt{Id}[t] \equiv \underline{\lambda}\mathtt{x}_1.\mathtt{x}_1 \\ & e_1 \,\Box\, e_2 \equiv \underline{\lambda}\mathtt{x}_1.e_1\underline{(}e_2\underline{(}\mathtt{x}_1\underline{)}\underline{)} \\ & \mathtt{Const}[t]\langle e\rangle \equiv \underline{\lambda}\mathtt{x}_1.e \\ & \mathtt{Fix}\ e \equiv \underline{\lambda}\mathtt{x}_1.\ \underline{\mathtt{fix}}\ e\underline{(}\mathtt{x}_1\underline{)} \\ & \mathtt{If}\langle e_1,e_2,e_3\rangle \equiv \underline{\lambda}\mathtt{x}_1.\ \underline{\mathtt{if}}\ e_1\underline{(}\mathtt{x}_1\underline{)}\ \underline{\mathtt{then}}\ e_2\underline{(}\mathtt{x}_1\underline{)}\ \underline{\mathtt{else}}\ e_3\underline{(}\mathtt{x}_1\underline{)} \end{aligned}$$

This should be rather familiar to anyone who knows a bit of categorical logic or a bit of a functional language like FP. In this notation the sequencing operator $\mathtt{seq}$ used above simply is

$$\mathtt{seq} = \lambda\mathtt{x}_1[\underline{\mathtt{Sto}{\rightarrow}\mathtt{Sto}}].\ \lambda\mathtt{x}_2[\underline{\mathtt{Sto}{\rightarrow}\mathtt{Sto}}].\ \mathtt{x}_2 \,\Box\, \mathtt{x}_1$$

$$tenv \vdash_{c1,c2} \mathtt{f}_{\mathrm{i}}[t] : t \qquad \text{if } \vdash_{c1} \; t$$

$$\frac{tenv \vdash_{c1,c2} e_1 : t_1 \qquad tenv \vdash_{c1,c2} e_2 : t_2}{tenv \vdash_{c1,c2} \langle e_1, e_2 \rangle : t_1 \times t_2}$$

$$\frac{tenv \vdash_{c1,c2} e_1 : t \underline{\rightarrow} t_1 \qquad tenv \vdash_{c1,c2} e_2 : t \underline{\rightarrow} t_2}{tenv \vdash_{c1,c2} \mathtt{Tuple}\langle e_1, e_2 \rangle : t \underline{\rightarrow} t_1 \underline{\times} t_2}$$

$$\text{if } \vdash_{c1} \; (t \underline{\rightarrow} t_1) \rightarrow (t \underline{\rightarrow} t_2) \rightarrow (t \underline{\rightarrow} t_1 \underline{\times} t_2)$$

$$\frac{tenv \vdash_{c1,c2} e : t_1 \times t_2}{tenv \vdash_{c1,c2} \mathtt{fst}\; e : t_1}$$

$$\frac{tenv \vdash_{c1,c2} e : t \underline{\rightarrow} t_1 \underline{\times} t_2}{tenv \vdash_{c1,c2} \mathtt{Fst}\; e : t \underline{\rightarrow} t_1}$$

$$\text{if } \vdash_{c1} \; (t \underline{\rightarrow} t_1 \underline{\times} t_2) \rightarrow (t \underline{\rightarrow} t_1)$$

$$\frac{tenv \vdash_{c1,c2} e : t_1 \times t_2}{tenv \vdash_{c1,c2} \mathtt{snd}\; e : t_2}$$

$$\frac{tenv \vdash_{c1,c2} e : t \underline{\rightarrow} t_1 \underline{\times} t_2}{tenv \vdash_{c1,c2} \mathtt{Snd}\; e : t \underline{\rightarrow} t_2}$$

$$\text{if } \vdash_{c1} \; (t \underline{\rightarrow} t_1 \underline{\times} t_2) \rightarrow (t \underline{\rightarrow} t_2)$$

$$\frac{tenv[t/\mathtt{x}_{\mathrm{i}}] \vdash_{c1,c2} e : t'}{tenv \vdash_{c1,c2} \lambda \mathtt{x}_{\mathrm{i}}[t].e : t \rightarrow t'} \quad \text{if } \vdash_{c2} \; t$$

$$\frac{tenv \vdash_{c1,c2} e : t \underline{\times} t' \underline{\rightarrow} t''}{tenv \vdash_{c1,c2} \mathtt{Curry}\; e : t \underline{\rightarrow} t' \underline{\rightarrow} t''}$$

$$\text{if } \vdash_{c1} \; (t \underline{\times} t' \underline{\rightarrow} t'') \rightarrow (t \underline{\rightarrow} t' \underline{\rightarrow} t'')$$

$$\frac{tenv \vdash_{c1,c2} e_1 : t' \rightarrow t \qquad tenv \vdash_{c1,c2} e_2 : t'}{tenv \vdash_{c1,c2} e_1(e_2) : t}$$

$$\frac{tenv \vdash_{c1,c2} e_1 : t \underline{\rightarrow} t' \underline{\rightarrow} t'' \qquad tenv \vdash_{c1,c2} e_2 : t \underline{\rightarrow} t'}{tenv \vdash_{c1,c2} \mathtt{Apply}\langle e_1, e_2 \rangle : t \underline{\rightarrow} t''}$$

$$\text{if } \vdash_{c1} \; (t \underline{\rightarrow} t' \underline{\rightarrow} t'') \rightarrow (t \underline{\rightarrow} t') \rightarrow t \underline{\rightarrow} t''$$

$$tenv \vdash_{c1,c2} \mathtt{x}_{\mathrm{i}} : t \qquad \text{if } \vdash_{c2} \; t \wedge tenv(\mathtt{x}_{\mathrm{i}}) = t$$

Table 1. Wellformedness of Expressions (part 1)

and thus there hardly is any need to name it.

To complete the definition of the two-level metalanguage we must explain when expressions are well-formed. We have already said that we shall need to impose conditions on the types as we go along and the well-formedness condition will be influenced by this although in a rather indirect way. As we shall see later these parameters may restrict types so that they e.g. denote complete lattices. We shall therefore write TML[**c1**,**c2**] for a

$$tenv \vdash_{c1,c2} \mathtt{Id}[t] : t\underline{\rightarrow}t \qquad \text{if } \vdash_{c1} t\underline{\rightarrow}t$$

$$\frac{tenv \vdash_{c1,c2} e_1 : t_0\underline{\rightarrow}t_1 \qquad tenv \vdash_{c1,c2} e_2 : t_1\underline{\rightarrow}t_2}{tenv \vdash_{c1,c2} e_2 \,\square\, e_1 : t_0\underline{\rightarrow}t_2}$$
$$\text{if } \vdash_{c1} (t_1\underline{\rightarrow}t_2) \rightarrow (t_0\underline{\rightarrow}t_1) \rightarrow (t_0\underline{\rightarrow}t_2)$$

$$\frac{tenv \vdash_{c1,c2} e : t'}{tenv \vdash_{c1,c2} \mathtt{Const}[t]\; e : t\underline{\rightarrow}t'}$$
$$\text{if } \vdash_{c1} t' \rightarrow t\underline{\rightarrow}t'$$

$$\frac{tenv \vdash_{c1,c2} e : t\rightarrow t}{tenv \vdash_{c1,c2} \mathtt{fix}\; e : t}$$

$$\frac{tenv \vdash_{c1,c2} e : t\underline{\rightarrow}t'\underline{\rightarrow}t'}{tenv \vdash_{c1,c2} \mathtt{Fix}\; e : t\underline{\rightarrow}t'}$$
$$\text{if } \vdash_{c1} (t\underline{\rightarrow}t'\underline{\rightarrow}t') \rightarrow (t\underline{\rightarrow}t')$$

$$\frac{tenv \vdash_{c1,c2} e_1 : \mathtt{Bool} \qquad tenv \vdash_{c1,c2} e_2 : t \qquad tenv \vdash_{c1,c2} e_3 : t}{tenv \vdash_{c1,c2} \mathtt{if}\; e_1\; \mathtt{then}\; e_2\; \mathtt{else}\; e_3 : t}$$

$$\frac{tenv \vdash_{c1,c2} e_1 : t\underline{\rightarrow\mathtt{Bool}} \qquad tenv \vdash_{c1,c2} e_2 : t\underline{\rightarrow}t' \qquad tenv \vdash_{c1,c2} e_3 : t\underline{\rightarrow}t'}{tenv \vdash_{c1,c2} \mathtt{If}\langle e_1,e_2,e_3\rangle : t\underline{\rightarrow}t'}$$
$$\text{if } \vdash_{c1} (t\underline{\rightarrow\mathtt{Bool}}) \rightarrow (t\underline{\rightarrow}t') \rightarrow (t\underline{\rightarrow}t') \rightarrow (t\underline{\rightarrow}t')$$

Table 2. Wellformedness of Expressions (part 2)

version of the two-level metalanguage where types are constrained as indicated by the parameters **c1** and **c2**. We then write $\vdash_c t$ to express that the type t is well-formed with respect to the constraint **c**. Whenever we say that t is a well-formed type of TML[**c1**,**c2**] we shall mean $\vdash_{c2} t$ because in general **c2** will be more liberal than **c1**, i.e. **c1** will imply **c2**. Next we write

$$tenv \vdash_{c1,c2} e : t$$

for the well-formedness of an expression e of intended type t assuming that the free variables of e have types as given by *tenv*. Here *tenv* is a type environment, i.e. a mapping from a finite subset of the variables $\{\mathbf{x}_i | i \in I\}$ to the types T. We refer to Tables 1 and 2 for the definition of $tenv \vdash_{c1,c2} e : t$ but we point out that the constraint **c2** is used to constrain the types of variables whereas the constraint **c1** is used to constrain the types of basic expressions and combinators.

We shall say that the expression e is *closed* if it has no free variables so that *tenv* may be taken as a mapping from the empty set. Also we shall say that a combinator ψ is *used with type* t_ψ, if $\vdash_{c1}\ t_\psi$ is the side condition that needs to be verified in order to apply the rule for ψ. As an example, $\Box$ is used with type $(\underline{\mathtt{Sto}\rightarrow\mathtt{Sto}})\rightarrow(\underline{\mathtt{Sto}\rightarrow\mathtt{Sto}})\rightarrow(\underline{\mathtt{Sto}\rightarrow\mathtt{Sto}})$ in the expression for `seq` displayed above.

Fact 3.1.4. If $tenv \vdash_{c1,c2}\ e : t_1$ and $tenv \vdash_{c1,c2}\ e : t_2$ then $t_1 = t_2$. $\Box$

3.1.3 Pragmatics of the metalanguage

We shall end this subsection with a few pragmatic considerations about the relationship between the two-level metalanguage and the typed λ-calculus we took as our starting point. One of our first points was not to pay great attention to the difference between a typed λ-calculus and an untyped λ-calculus because the various algorithms for *type analysis* might be of use in transferring types into an otherwise untyped expression. In quite an analogous way we shall not pay great attention to the difference between a typed λ-calculus and a two-level λ-calculus as one can develop an algorithm for *binding time analysis* [Nielson, 1988b] that is useful for transferring the underlining distinction into a typed expression without this distinction. Continuing this line of argument we shall not pay great attention to the difference between a two-level λ-calculus and the two-level metalanguage adopted in Definition 3.1.3 because one can develop a variant of bracket abstraction (called *two-level λ-lifting* [Nielson, 1988c]) that will aid in transforming underlined constructs to combinator form.

The choice of combinators in the two-level metalanguage suits the λ-calculus well but one may regard them as nothing but glorified versions of the basic expressions $\mathtt{f}_\mathrm{i}[t]$, i.e. that for a few of the basic expressions $\mathtt{f}_\mathrm{i}[t]$ we have decided to use a different syntax. This means that one could as well study combinator-like basic expressions that would be more suitable for languages like PASCAL, PROLOG, OCCAM or *action semantics*. However, we always have the λ-notation available and we would only wish to restrict this in settings where the resulting metalanguage is so big as to make it hard to develop an analysis. We shall see examples of this in the next subsection where we define the parameterised semantics of the metalanguage.

3.2 Specification of Analyses

Following most approaches to denotational semantics we shall interpret the types of the metalanguage as *domains*. We saw in Section 2 that for abstract interpretation there is a special interest in the *complete lattices* and we shall restrict our attention to the *algebraic lattices* which are those complete lattices that are additionally domains. Roughly the idea will be to interpret the non-underlined type constructs as domains whereas underlined type constructs will be interpreted as algebraic lattices when we are specifying abstract interpretations and as domains when we are specifying the standard interpretation.

To be selfcontained we shall briefly review a few concepts that have been treated at greater length in previous chapters of this handbook.

Definition 3.2.1. A *chain* in a partially ordered set $D=(D,\sqsubseteq)$ is a sequence $(d_n)_n$ of elements indexed by the natural numbers such that $d_n \sqsubseteq d_m$ whenever $n \leq m$. A *cpo* D is a partially ordered set with a least element, $\perp$, and in which every chain $(d_n)_n$ has a (necessarily unique) least upper bound, $\bigsqcup_n d_n$. The cpo D is consistently complete if every subset Y of D that has an upper bound in D also has a least upper bound $\bigsqcup Y$ in D. An element b in a cpo D is *compact* if whenever $b \sqsubseteq \bigsqcup_n d_n$ for a chain $(d_n)_n$ we have some natural number n such that $b \sqsubseteq d_n$. A subset B of D is a *basis* if every element d of D can be written as $d = \bigsqcup_n b_n$ where $(b_n)_n$ is a chain in D with each b_n an element of B. A *domain* is a consistently complete cpo with a countable basis B_D of compact elements. We shall use the term *algebraic lattice* for those complete lattices that are also domains, i.e. for those domains in which any subset has an upper bound. □

Definition 3.2.2. A function $f{:}D \rightarrow E$ from a domain $D=(D,\sqsubseteq)$ to another domain $E=(E,\sqsubseteq)$ is *monotonic* if it preserves the partial order and is *continuous* if it preserves the least upper bounds of chains, i.e. $f(\bigsqcup_n d_n) = \bigsqcup_n f(d_n)$. It is *additive* (sometimes called linear) if it preserves all least upper bounds, i.e. $f(\bigsqcup Y) = \bigsqcup\{f(y) \mid y \in Y\}$ whenever Y has a least upper bound. It is *binary additive* if $f(d_1 \sqcup d_2) = f(d_1) \sqcup f(d_2)$ and is *strict* if it preserves the least element, i.e. $f(\perp)=\perp$. It is *compact preserving* if it preserves compact elements, i.e. $f(b) \in B_E$ whenever $b \in B_D$. A continuous function $f{:}D \rightarrow D$ from a domain $D=(D,\sqsubseteq)$ to itself has a least fixed point given by $\mathrm{FIX}(f) = \bigsqcup_n f^n(\perp)$, i.e. $f(\mathrm{FIX}(f)) = \mathrm{FIX}(f)$ and whenever $f(d) = d$ (or indeed $f(d) \sqsubseteq d$) we have $\mathrm{FIX}(f) \sqsubseteq d$. □

Definition 3.2.3. A predicate P over a domain $D=(D,\sqsubseteq)$ is a function from D to the set {true,false} of truth values. It is *admissible* if $P(\perp)$ holds and if $P(\bigsqcup_n d_n)$ holds whenever $(d_n)_n$ is a chain such that $P(d_n)$ holds for every element d_n. For an admissible predicate P we have the induction principle

$$\frac{P(d) \Rightarrow P(f(d))}{P(\mathrm{FIX}(f))}$$

whenever f is continuous. In a similar way we define the notion of admissible relation since a relation between the domains $D_1,\ldots,D_n$ $(n \geq 1)$ is nothing but a predicate over the cartesian product $D_1 \times \ldots \times D_n$ (where the partial order is given in the usual componentwise manner). □

3.2.1 Interpreting the types

For a type t the definition of its interpretation $[\![t]\!](\mathcal{I})$ is by structure on the syntax of t. As we shall see it will make use of the parameter $\mathcal{I}$ whenever underlined constructs are encountered. Actually, the parameter $\mathcal{I}$ may be regarded as being a pair $(\mathcal{I}^{\mathrm{t}},\mathcal{I}^{\mathrm{e}})$ and for the definition of $[\![t]\!](\mathcal{I})$ it is only the $\mathcal{I}^{\mathrm{t}}$ component that will be needed.

$$[\![\ \mathtt{A}_{\mathrm{i}}\ ]\!](\mathcal{I}) = \text{some a priori specified domain } A_{\mathrm{i}}$$
with A_{bool} the domain {true,false,$\perp$} of booleans
and A_{int} the domain {...,-1,0,1,...,$\perp$} of integers

$$[\![\ t_1{\times}t_2\ ]\!](\mathcal{I}) = [\![\ t_1\ ]\!](\mathcal{I}) \times [\![\ t_2\ ]\!](\mathcal{I})$$
where the elements are the pairs of elements
and the partial order is defined componentwise

$$[\![\ t_1{\rightarrow}t_2\ ]\!](\mathcal{I}) = [\![\ t_1\ ]\!](\mathcal{I}) \rightarrow [\![\ t_2\ ]\!](\mathcal{I})$$
where the elements are the continuous functions
and the partial order is $f \sqsubseteq g$ iff $\forall d$: $f(d) \sqsubseteq g(d)$

$$[\![\ \underline{\mathtt{A}}_{\mathrm{i}}\ ]\!](\mathcal{I}) = \mathcal{I}^{\mathrm{t}}_{\mathrm{i}}$$

$$[\![\ t_1 \underline{\times}\ t_2\ ]\!](\mathcal{I}) = \mathcal{I}^{\mathrm{t}}_{\times}([\![t_1]\!](\mathcal{I}),[\![t_2]\!](\mathcal{I}))$$

$$[\![\ t_1 \underline{\rightarrow}\ t_2\ ]\!](\mathcal{I}) = \mathcal{I}^{\mathrm{t}}_{\rightarrow}([\![t_1]\!](\mathcal{I}),[\![t_2]\!](\mathcal{I}))$$

The demands on the parameter $\mathcal{I}$ are expressed in

Definition 3.2.4. An *interpretation* $\mathcal{I}$ (or $\mathcal{I}^{\mathrm{t}}$) *of types* is a specification of

- a property $\mathcal{I}^{\mathrm{t}}_{\mathrm{P}} = \mathbf{P}$ of domains (e.g. 'is a domain' or 'is an algebraic lattice'),
- for each i a domain $\mathcal{I}^{\mathrm{t}}_{\mathrm{i}}$ with property $\mathbf{P}$,
- operations $\mathcal{I}^{\mathrm{t}}_{\times}$ and $\mathcal{I}^{\mathrm{t}}_{\rightarrow}$ on domains with property $\mathbf{P}$ such that the result is a domain with property $\mathbf{P}$.

We shall use the term *domain interpretation* for an interpretation of types where the property $\mathbf{P}$ equals 'is a domain' and we shall use the term *lattice interpretation* for an interpretation of types where the property $\mathbf{P}$ equals 'is an algebraic lattice'. □

Clearly domain interpretations are of relevance when specifying a standard semantics and lattice interpretations are of relevance when specifying abstract interpretations.

Unfortunately we will have to impose certain well-formedness conditions upon types for the above equations to define a domain. As an example, $\texttt{Int}\underline{\times}\underline{\texttt{Int}}$ will not be well-formed because

$$[\![\ \texttt{Int}\underline{\times}\underline{\texttt{Int}}\ ]\!](\mathcal{I}) = \mathcal{I}^{\mathrm{t}}_{\underline{\times}}(A_{\mathrm{int}}, \mathcal{I}^{\mathrm{t}}_{\underline{\mathrm{int}}})$$

and even though $\mathcal{I}^{\mathrm{t}}_{\underline{\mathrm{int}}}$ is an algebraic lattice (e.g. that for the detection of signs), A_{int} is not and so one cannot apply $\mathcal{I}^{\mathrm{t}}_{\underline{\times}}$ when $\mathcal{I}$ is a lattice interpretation. Since we have argued that the use of (algebraic) lattices is a very natural setup for abstract interpretation we conclude that we should ban $\texttt{Int}\underline{\times}\underline{\texttt{Int}}$.

With this motivation we shall define the predicates

lt(t) to ensure that t will be interpreted as an algebraic lattice when performing abstract interpretation,

dt(t) to ensure that t will be interpreted as a domain in any interpretation.

Definition 3.2.5. The predicates **lt** (for lattice type) and **dt** (for domain type) are defined by

	$\texttt{A}_{\mathrm{i}}$	$\underline{\texttt{A}}_{\mathrm{i}}$	$t_1{\times}t_2$	$t_1\underline{\times}t_2$	$t_1{\rightarrow}t_2$	$t_1\underline{\rightarrow}t_2$
lt	false	true	$\mathbf{lt}_1{\wedge}\mathbf{lt}_2$	$\mathbf{lt}_1{\wedge}\mathbf{lt}_2$	$\mathbf{dt}_1{\wedge}\mathbf{lt}_2$	$\mathbf{lt}_1{\wedge}\mathbf{lt}_2$
dt	true	true	$\mathbf{dt}_1{\wedge}\mathbf{dt}_2$	$\mathbf{lt}_1{\wedge}\mathbf{lt}_2$	$\mathbf{dt}_1{\wedge}\mathbf{dt}_2$	$\mathbf{lt}_1{\wedge}\mathbf{lt}_2$

where we write $\mathbf{lt}_1$ for $\mathbf{lt}(t_1)$ etc. □

We shall regard a type t as being well-formed whenever $\mathbf{dt}(t)$ holds and write $\vdash_{dt}\ t$ as a record of this.

Proposition 3.2.6. The equations for $[\![t]\!](\mathcal{I})$ define a domain when t is a well-formed type in TML[**dt**,**dt**] and $\mathcal{I}$ is a domain or lattice interpretation. □

Proof: Let $\mathcal{I}$ be an interpretation of types that specifies the property $\mathcal{I}^{\mathrm{t}}_{\mathrm{P}} = \mathbf{P}$ where $\mathbf{P}(D)$ either means that D is a domain or that D is an algebraic lattice. By induction on the structure of types t we will show

- if $\mathbf{dt}(t)$ then $[\![t]\!](\mathcal{I})$ specifies a domain,
- if $\mathbf{lt}(t)$ then $\mathbf{dt}(t)$ and $[\![t]\!](\mathcal{I})$ has property $\mathbf{P}$.

The cases $\texttt{A}_{\mathrm{i}}$ and $\underline{\texttt{A}}_{\mathrm{i}}$ are straightforward. The case $t_1{\times}t_2$ follows because $D_1{\times}D_2$ is a domain whenever D_1 and D_2 are and an algebraic lattice whenever D_1 and D_2 are. The case $t_1\underline{\times}t_2$ follows from the assumptions. The case $t_1 \rightarrow t_2$ follows because $D_1 \rightarrow D_2$ is a domain when D_1 and D_2 are and an algebraic lattice when D_1 is a domain and D_2 is an algebraic lattice. The case $t_1 \underline{\rightarrow} t_2$ follows from the assumptions. □

We thus see that a type like $\mathtt{Int}\underline{\times}\underline{\mathtt{Int}}$ is not well-formed whereas (generalising [Nielson, 1989]) a type like $(\mathtt{Int}\rightarrow\underline{\mathtt{Int}})\underline{\times}\underline{\mathtt{Int}}$ will be well-formed and will denote an algebraic lattice in any abstract interpretation (i.e. in any lattice interpretation). Furthermore it should now be clear that $\mathtt{A}_{\mathrm{int}\times\mathrm{bool}}$, $\mathtt{A}_{\mathrm{int}}\times\mathtt{A}_{\mathrm{bool}}$, $\underline{\mathtt{A}}_{\mathrm{int}\times\mathrm{bool}}$ and $\underline{\mathtt{A}}_{\mathrm{int}}\underline{\times}\underline{\mathtt{A}}_{\mathrm{bool}}$ will be treated differently in the semantics.

3.2.2 Interpreting the expressions

To define the meaning of a well-formed expression we shall consider a type environment $tenv$ with domain $\{\mathtt{x}_1,\ldots,\mathtt{x}_\mathrm{n}\}$[2] and a well-formed expression e of type t, i.e.

$$tenv \vdash_{dt,dt} \; e : t$$

Without loss of generality we may assume that $\vdash_{dt} \; t_\mathrm{i}$ whenever $t_\mathrm{i} = tenv(\mathtt{x}_\mathrm{i})$ as otherwise $\mathtt{x}_\mathrm{i}$ could be removed from the type environment (due to the formulation of the axiom for $\mathtt{x}_\mathrm{i}$ in Table 1). The semantics of e relative to the interpretation $\mathcal{I}$ is an entity

$$[\![\; e \;]\!]_{\mathrm{tenv}}(\mathcal{I}) \in [\![t_1]\!](\mathcal{I})\times\ldots\times[\![t_\mathrm{n}]\!](\mathcal{I})\rightarrow[\![t]\!](\mathcal{I})$$

where again $t_\mathrm{i} = tenv(\mathtt{x}_\mathrm{i})$. That this makes sense is a consequence of

Fact 3.2.7. If $tenv \vdash_{dt,dt} \; e : t$ then $\vdash_{dt} \; t$. □

The definition of $[\![e]\!]_{\mathrm{tenv}}(\mathcal{I})$ is by structural induction on e and again we shall use the interpretation $\mathcal{I}$, i.e. $(\mathcal{I}^\mathrm{t},\mathcal{I}^\mathrm{e})$, supplied as a parameter when we come to the underlined constructs. Writing $[\![e]\!]\mathcal{I}\rho$ for $[\![e]\!]_{\mathrm{tenv}}(\mathcal{I})(\rho)$ we have

$[\![\; \mathtt{f}_\mathrm{i}[t] \;]\!]\mathcal{I}\rho = \mathcal{I}^\mathrm{e}_{\mathrm{i}[t]}$
$[\![\; \langle e_1,e_2\rangle \;]\!]\mathcal{I}\rho = ([\![e_1]\!]\mathcal{I}\rho,[\![e_2]\!]\mathcal{I}\rho)$
$[\![\; \underline{\mathtt{Tuple}}\langle e_1,e_2\rangle \;]\!]\mathcal{I}\rho = \mathcal{I}^\mathrm{e}_{\mathrm{Tuple}[t]}([\![e_1]\!]\mathcal{I}\rho)([\![e_2]\!]\mathcal{I}\rho)$ where $\mathtt{Tuple}$ is used with type t
$[\![\; \mathtt{fst} \; e \;]\!]\mathcal{I}\rho = d_1$ where $(d_1,d_2) = [\![e]\!]\mathcal{I}\rho$
$[\![\; \underline{\mathtt{Fst}} \; e \;]\!]\mathcal{I}\rho = \mathcal{I}^\mathrm{e}_{\mathrm{Fst}[t]}([\![e]\!]\mathcal{I}\rho)$ where $\mathtt{Fst}$ is used with type t
$[\![\; \mathtt{snd} \; e \;]\!]\mathcal{I}\rho = d_2$ where $(d_1,d_2) = [\![e]\!]\mathcal{I}\rho$
$[\![\; \underline{\mathtt{Snd}} \; e \;]\!]\mathcal{I}\rho = \mathcal{I}^\mathrm{e}_{\mathrm{Snd}[t]}([\![e]\!]\mathcal{I}\rho)$ where $\mathtt{Snd}$ is used with type t

[2] It is rather demanding to assume that dom($tenv$) is always of the form $\{\mathtt{x}_1,\ldots,\mathtt{x}_\mathrm{n}\}$ for some natural number n, but as it simplifies the notation considerably we shall stick to this assumption. Alternatively, one might model a type environment as a list of pairs of the form $(\mathtt{x}_\mathrm{i},t_\mathrm{i})$.

$[\![\ \lambda\mathtt{x}_{\mathrm{i}}[t].e\ ]\!]\mathcal{I}\rho = \lambda d{\in}[\![t]\!](\mathcal{I}).\ [\![e]\!]_{\mathrm{tenv}[t/\mathtt{x}_i]}(\mathcal{I})(\rho[d/\mathtt{x}_{\mathrm{i}}])$

where $(d_1,...,d_{\mathrm{n}})[d/\mathtt{x}_{\mathrm{i}}] = (d_1,...,d,...,d_{\mathrm{n}})$ if $\mathrm{i}{\leq}\mathrm{n}$

and $(d_1,...,d_{\mathrm{n}})[d/\mathtt{x}_{\mathrm{i}}] = (d_1,...,d_{\mathrm{n}},d)$ if $\mathrm{i}{=}\mathrm{n}{+}1$

$[\![\ \mathtt{Curry}\ e\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\mathrm{Curry}[t]}([\![e]\!]\mathcal{I}\rho)$ where **Curry** is used with type t
$[\![\ e_1(e_2)\ ]\!]\mathcal{I}\rho = ([\![e_1]\!]\mathcal{I}\rho)([\![e_2]\!]\mathcal{I}\rho)$
$[\![\ \mathtt{Apply}\langle e_1,e_2\rangle\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\mathrm{Apply}[t]}([\![e_1]\!]\mathcal{I}\rho)([\![e_2]\!]\mathcal{I}\rho)$ where **Apply** is used with type t
$[\![\ \mathtt{x}_{\mathrm{i}}\ ]\!]\mathcal{I}\rho = d_{\mathrm{i}}$ where $(d_1,...,d_{\mathrm{n}}) = \rho$
$[\![\ \mathtt{Id}[t]\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\mathrm{Id}[t \underline{\to} t]}$
$[\![\ e_1\ \Box\ e_2\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\Box[t]}([\![e_1]\!]\mathcal{I}\rho)([\![e_2]\!]\mathcal{I}\rho)$ where $\Box$ is used with type t
$[\![\ \mathtt{Const}[t]\ e\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\mathrm{Const}[t']}([\![e]\!]\mathcal{I}\rho)$ where $\mathtt{Const}[t]$ is of type t'
$[\![\ \mathtt{fix}\ e\ ]\!]\mathcal{I}\rho = \mathrm{FIX}([\![e]\!]\mathcal{I}\rho)$
$[\![\ \mathtt{Fix}\ e\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\mathrm{Fix}[t]}([\![e]\!]\mathcal{I}\rho)$ where **Fix** is used with type t

$$[\![\ \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3\ ]\!]\mathcal{I}\rho = \begin{cases} [\![e_2]\!]\mathcal{I}\rho & \text{if } [\![e_1]\!]\mathcal{I}\rho{=}\text{true} \\ [\![e_3]\!]\mathcal{I}\rho & \text{if } [\![e_1]\!]\mathcal{I}\rho{=}\text{false} \\ \bot & \text{if } [\![e_1]\!]\mathcal{I}\rho{=}\bot \end{cases}$$

$[\![\ \mathtt{If}\langle e_1,e_2,e_3\rangle\ ]\!]\mathcal{I}\rho = \mathcal{I}^{\mathrm{e}}_{\mathrm{If}[t]}([\![e_1]\!]\mathcal{I}\rho)([\![e_2]\!]\mathcal{I}\rho)([\![e_3]\!]\mathcal{I}\rho)$ where **If** is used with type t

To prevent any misconception we point out that the pattern matching, e.g.

d_1 where $(d_1,d_2) = [\![e]\!]\rho\mathcal{I}$

may be replaced by the use of explicit destructors, e.g.

$p{\downarrow}1$ where $p = [\![e]\!]\rho\mathcal{I}$

and that similarly the 'where' may be replaced by textual substitution, e.g.

$([\![e]\!]\rho\mathcal{I}){\downarrow}1$

The demands on the parameter $\mathcal{I}$ are clarified by:

Definition 3.2.8. An interpretation $\mathcal{I}$ is a specification of

- an interpretation $\mathcal{I}^{\mathrm{t}}$ of types that is a domain interpretation or a lattice interpretation,
- for each basic expression or combinator an entity in the required domain, i.e.

$$\mathcal{I}^{\mathrm{e}}_{\mathrm{i}[t]} \in \llbracket t \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Tuple}[(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t'\underline{\times}t'')]} \in \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t'\underline{\times}t'' \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Fst}[(t\underline{\rightarrow}t'\underline{\times}t'')\rightarrow(t\underline{\rightarrow}t')]} \in \llbracket t\underline{\rightarrow}t'\underline{\times}t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Snd}[(t\underline{\rightarrow}t'\underline{\times}t'')\rightarrow(t\underline{\rightarrow}t'')]} \in \llbracket t\underline{\rightarrow}t'\underline{\times}t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t'' \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Curry}[(t'\underline{\times}t''\underline{\rightarrow}t)\rightarrow(t'\underline{\rightarrow}t''\underline{\rightarrow}t)]} \in \llbracket t'\underline{\times}t''\underline{\rightarrow}t \rrbracket(\mathcal{I}) \rightarrow \llbracket t'\underline{\rightarrow}t''\underline{\rightarrow}t \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Apply}[(t\underline{\rightarrow}t'\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')]} \in \llbracket t\underline{\rightarrow}t'\underline{\rightarrow}t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t'' \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Id}[t\underline{\rightarrow}t]} \in \llbracket t\underline{\rightarrow}t \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\Box[(t'\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')]} \in \llbracket t'\underline{\rightarrow}t'' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t'' \rrbracket(\mathcal{I})$$

$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Const}[t'\rightarrow t\underline{\rightarrow}t']} \in \llbracket t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{Fix}[(t\underline{\rightarrow}t'\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]} \in \llbracket t\underline{\rightarrow}t'\underline{\rightarrow}t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I})$$
$$\mathcal{I}^{\mathrm{e}}_{\mathrm{If}[(t\underline{\rightarrow Bool})\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]} \in \llbracket t\underline{\rightarrow\mathtt{Bool}} \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I}) \rightarrow \llbracket t\underline{\rightarrow}t' \rrbracket(\mathcal{I})$$

Here we must assume that the types t indexing each $\mathcal{I}^{\mathrm{e}}_{\psi[t]}$ are such that $\llbracket\cdots\rrbracket(\mathcal{I})$ is only applied to well-formed types, i.e. types t' such that $\vdash_{dt}\ t'$, and this is equivalent to assuming that $\vdash_{dt}\ t$.

To simplify the notation we shall henceforth feel free to omit the type and type environments as subscripts and thus write $\llbracket e \rrbracket(\mathcal{I})$ for $\llbracket e \rrbracket_{\mathrm{tenv}}(\mathcal{I})$ and $\mathcal{I}^{\mathrm{e}}_{\psi}$ for $\mathcal{I}^{\mathrm{e}}_{\psi[t]}$. (In a sense we regard the combinators as having a kind of polymorphic interpretation.)

Proposition 3.2.9. The equations for $\llbracket e \rrbracket(\mathcal{I})$ define a value when e is a well-formed expression in TML[**dt**,**dt**] and $\mathcal{I}$ is a domain or lattice interpretation. $\Box$

Proof: The assumptions on $\mathcal{I}$ ensure that $\llbracket t \rrbracket(\mathcal{I})$ is a well-defined domain whenever $\vdash_{dt}\ t$. We shall show by structural induction on an expression e that

> if $tenv \vdash_{dt,dt}\ e : t$ where $\mathrm{dom}(tenv) = \{\mathbf{x}_1,\ldots,\mathbf{x}_n\}$, $tenv(\mathbf{x}_i) = t_i$ and $\vdash_{dt}\ t_i$
> then $\llbracket e \rrbracket_{\mathrm{tenv}}(\mathcal{I}) \in \llbracket t_1 \rrbracket(\mathcal{I})\times\ldots\times\llbracket t_n \rrbracket(\mathcal{I})\rightarrow\llbracket t \rrbracket(\mathcal{I})$ and this domain does exist.

The proof makes use of Fact 3.1.4 to ensure that the type t of e is unique so that also the types indexing each $\mathcal{I}^{\mathrm{e}}_{\psi}$ are unique. Furthermore it makes use of Fact 3.2.7 to ensure that the type t of e is well-formed so that, given the inference rules of Tables 1 and 2, we only request a $\mathcal{I}^{\mathrm{e}}_{\psi[t]}$ in a domain that must exist by Proposition 3.2.6. The structural induction is now mostly straightforward and we shall omit the details. □

3.2.3 Example interpretations

We now present a total of five examples: a lazy standard semantics, detection of signs (in an independent attribute formulation), strictness, liveness, and detection of signs (in a relational formulation). The main point of these examples is to demonstrate the generality obtained by varying the interpretation of the underlined types and type constructors. The last two examples are somewhat technical and the details are not vital for the remainder of the development.

Example 3.2.10. (Lazy Standard Semantics) In this example we define the standard semantics of the metalanguage. This amounts to specifying a domain interpretation $\mathbf{S}$ and for types we have:

- the property $\mathbf{S}^{\mathrm{t}}_{\mathrm{P}}$ equals 'is a domain',
- $\mathbf{S}^{\mathrm{t}}_{\mathrm{i}} = A_{\mathrm{i}}$ (the a priori chosen domains for the types $\mathtt{A}_{\mathrm{i}}$),
- $\mathbf{S}^{\mathrm{t}}_{\underline{\times}} = \times$ (cartesian product) and $\mathbf{S}^{\mathrm{t}}_{\underline{\rightarrow}} = \rightarrow$ (continuous function space).

In other words we do not distinguish between underlined and non-underlined types and constructors and this should not be surprising in a *standard semantics*. (We may note from this example that well-formedness of a type t is a sufficient condition for $[\![t]\!](\mathbf{S})$ to be defined but it is not necessary as $[\![t]\!](\mathbf{S})$ is in fact defined as a domain for all types t.) If we wanted an *eager* standard semantics instead we might take $\mathbf{S}^{\mathrm{t}}_{\underline{\times}}$ to be a so-called *smash* product and $\mathbf{S}^{\mathrm{t}}_{\underline{\rightarrow}}$ to be *strict* function space.

Turning to the expression part we have:

$\mathbf{S}^{\mathrm{e}}_{\mathrm{i}[t]}$ is some a priori fixed element of $[\![t]\!](\mathbf{S})$
 e.g. $\mathbf{S}^{\mathrm{e}}_{\mathrm{true}} = \mathrm{true}$ etc.
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Tuple}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ (v_1(w), v_2(w))$
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Fst}} = \lambda v.\ \lambda w.\ d_1 \text{ where } (d_1,d_2){=}v(w)$
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Snd}} = \lambda v.\ \lambda w.\ d_2 \text{ where } (d_1,d_2){=}v(w)$
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Curry}} = \lambda v.\ \lambda w_1.\ \lambda w_2.\ v(w_1,w_2)$
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Apply}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(w)(v_2(w))$
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Id}} = \lambda w.w$
$\mathbf{S}^{\mathrm{e}}_{\underline{\square}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$
$\mathbf{S}^{\mathrm{e}}_{\mathrm{Const}} = \lambda v.\ \lambda w.\ v$

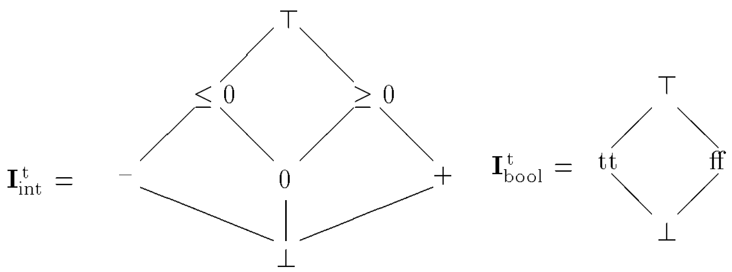
$$\mathbf{S}^{\mathrm{e}}_{\mathrm{Fix}} = \lambda v.\ \lambda w.\ \mathrm{FIX}(v(w))$$

$$\mathbf{S}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ \begin{cases} v_2(w) & \text{if } v_1(w)\text{=true} \\ v_3(w) & \text{if } v_1(w)\text{=false} \\ \perp & \text{if } v_1(w)\text{=}\perp \end{cases}$$

This definition is in agreement with the informal explanation of the combinators that we gave in Subsection 3.1. □

Example 3.2.11. (Detection of Signs — **I**) In this example we do need the distinction between underlined and non-underlined types in order to be able to formalize the abstract interpretation for detecting the signs of the integers. We specify a lattice interpretation **I** and for types we have:

- the property $\mathbf{I}^{\mathrm{t}}_{\mathrm{P}}$ equals 'is an algebraic lattice',
- the lattices $\mathbf{I}^{\mathrm{t}}_{\mathrm{i}}$ include



- $\mathbf{I}^{\mathrm{t}}_{\times} = \times$ (cartesian product) and $\mathbf{I}^{\mathrm{t}}_{\underline{\rightarrow}} = \rightarrow$ (continuous function space).

Turning to the expressions we have:

$\mathbf{I}^{\mathrm{e}}_{\mathrm{i}[t]}$ is some a priori fixed element of $[\![t]\!](\mathbf{I})$
e.g. $\mathbf{I}^{\mathrm{e}}_{\mathrm{true}} = \mathrm{tt}$, $\mathbf{I}^{\mathrm{e}}_{1} = +$ etc.

$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Tuple}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ (v_1(w),v_2(w))$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Fst}} = \lambda v.\ \lambda w.\ d_1 \text{ where } (d_1,d_2)=v(w)$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Snd}} = \lambda v.\ \lambda w.\ d_2 \text{ where } (d_1,d_2)=v(w)$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Curry}} = \lambda v.\ \lambda w_1.\ \lambda w_2.\ v(w_1,w_2)$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Apply}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(w)(v_2(w))$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Id}} = \lambda w.w$$
$$\mathbf{I}^{\mathrm{e}}_{\Box} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Const}} = \lambda v.\ \lambda w.\ v$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{Fix}} = \lambda v.\ \lambda w.\ \mathrm{FIX}(v(w))$$
$$\mathbf{I}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ \begin{cases} v_2(w) & \text{if } v_1(w)\text{=tt} \\ v_3(w) & \text{if } v_1(w)\text{=ff} \\ \perp & \text{if } v_1(w)\text{=}\perp \\ v_2(w)\sqcup v_3(w) & \text{if } v_1(w)\text{=}\top \end{cases}$$

To see that $\mathbf{I}^{\mathrm{e}}_{\mathrm{If}[(t\underline{\rightarrow Bool})\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]}$ is well-defined we shall assume that the type in the subscript is well-formed. Then we have $\mathbf{lt}(t')$ so that by (the proof of) Proposition 3.2.6 the domain $[\![t']\!](\mathbf{I})$ is a complete lattice. Hence the least upper bound exists and as the binary least upper bound operation is continuous, $\mathbf{I}^{\mathrm{e}}_{\mathrm{If}}$ will be an element of the required (continuous) function space. $\square$

Example 3.2.12. (Strictness) Simplifying the lattices of the previous example we arrive at a strictness analysis. Since this analysis is by far the most cited analysis for lazy functional languages we briefly present its specification. As in the previous example we specify a lattice interpretation $\mathbf{T}$ and for types we have:

- the property $\mathbf{T}^{\mathrm{t}}_{\mathrm{P}}$ equals 'is an algebraic lattice',
- the lattices $\mathbf{T}^{\mathrm{t}}_{\mathrm{i}}$ are

  $$\mathbf{T}^{\mathrm{t}}_{\mathrm{i}} = \begin{array}{l} \bullet\ \mathit{1} \\ \mid \\ \bullet\ \mathit{0} \end{array}$$

- $\mathbf{T}^{\mathrm{t}}_{\times} = \times$ (cartesian product) and $\mathbf{T}^{\mathrm{t}}_{\rightarrow} = \rightarrow$ (continuous function space).

Turning to the expressions we have:

$\mathbf{T}^{\mathrm{e}}_{\mathrm{i}[t]}$ is some a priori fixed element of $[\![t]\!](\mathbf{T})$
  e.g. $\mathbf{T}^{\mathrm{e}}_{\mathrm{true}} = \mathit{1}$, $\mathbf{T}^{\mathrm{e}}_{1} = \mathit{1}$ etc.

$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Tuple}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ (v_1(w), v_2(w))$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Fst}} = \lambda v.\ \lambda w.\ d_1 \text{ where } (d_1,d_2)=v(w)$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Snd}} = \lambda v.\ \lambda w.\ d_2 \text{ where } (d_1,d_2)=v(w)$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Curry}} = \lambda v.\ \lambda w_1.\ \lambda w_2.\ v(w_1,w_2)$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Apply}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(w)(v_2(w))$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Id}} = \lambda w.w$$
$$\mathbf{T}^{\mathrm{e}}_{\Box} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Const}} = \lambda v.\ \lambda w.\ v$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{Fix}} = \lambda v.\ \lambda w.\ \mathrm{FIX}(v(w))$$
$$\mathbf{T}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ \begin{cases} \bot & \text{if } v_1(w)=\mathit{0} \\ v_2(w)\sqcup v_3(w) & \text{if } v_1(w)=\mathit{1} \end{cases}$$

Well-definedness of this specification follows much as in the previous example. $\square$

Example 3.2.13. (Liveness) In the previous two examples we used the ability to interpret the $\underline{\mathtt{A}}_{\mathrm{i}}$ in a different manner than in the standard semantics. In this example we shall additionally need the ability to interpret the type constructor $\underline{\rightarrow}$ in a different manner than in the standard semantics. The reason for this is that *liveness analysis* is a backward analysis which means that the direction of the analysis is opposite to the flow of control. We specify the analysis by defining a lattice interpretation $\mathbf{L}$ and for types we have:

- the property $\mathbf{L}^{\mathrm{t}}_{\mathrm{P}}$ equals 'is an algebraic lattice',
- the lattices $\mathbf{L}^{\mathrm{t}}_{\mathrm{i}}$ are

$$\mathbf{L}^{\mathrm{t}}_{\mathrm{i}} = \begin{array}{l} \bullet\, \mathit{live} \\ | \\ \bullet\, \mathit{dead} \end{array}$$

- the operators are $\mathbf{L}^{\mathrm{t}}_{\underline{\times}} = \times$ (cartesian product) and $\mathbf{L}^{\mathrm{t}}_{\underline{\rightarrow}} = \leftarrow$, i.e. $\mathbf{L}^{\mathrm{t}}_{\underline{\rightarrow}}(D,E) = D \leftarrow E = E \rightarrow D$ which is the domain of continuous functions from E to D.

Intuitively, *dead* means "will never be used later in any computation", while *live* means "may be used later in some computation". It might be argued that the analysis should be called a "deadness" analysis because it is the property *dead* that can be trusted; however, it is common terminology to use the term "liveness" analysis. We should also point out that a backwards liveness analysis for flowchart programs has been seen before (in section 2.5.3).

Note that the backward nature of the analysis is recorded by interpreting $\underline{\rightarrow}$ as $\leftarrow$ just as the forward nature of an analysis is recorded by interpreting $\underline{\rightarrow}$ as $\rightarrow$ (as in the previous example).

Turning to expressions we shall impose additional constraints on the types that these are allowed to have. The motivation is that liveness analyses usually are developed for flowchart languages only and here we do not wish to give a more encompassing definition. Doing so is indeed a hard research problem as it seems to involve mixing forward and backward components into one analysis; hence interpreting $\underline{\rightarrow}$ as $\leftarrow$ is likely to be too simple-minded in the general case [Hughes, 1988,Ammann, 1994].

Definition The predicates **sr** and **sc** are defined by[3]

	$\mathtt{A}_{\mathrm{i}}$	$\underline{\mathtt{A}}_{\mathrm{i}}$	$t_1 \times t_2$	$t_1 \underline{\times} t_2$	$t_1 \rightarrow t_2$	$t_1 \underline{\rightarrow} t_2$
sr	false	true	false	$\mathbf{sr}_1 \wedge \mathbf{sr}_2$	false	false
sc	true	false	$\mathbf{sc}_1 \wedge \mathbf{sc}_2$	false	$\mathbf{sc}_1 \wedge \mathbf{sc}_2$	$\mathbf{sr}_1 \wedge \mathbf{sr}_2$

[3]These acronyms relate to previous papers by one of the authors and **sr** stands for 'run-time types in TML_s' whereas **sc** stands for 'compile-time types in TML_s'.

The intention with $\mathbf{sr}(t)$ is that t is an all-underlined product of base types and the intention with $\mathbf{sc}(t)$ is that t only contains underlined constructs if these constructs are parts of an all-underlined type with just one function space constructor in it. Clearly $\mathbf{sr}(t)$ implies $\mathbf{lt}(t)$ and hence $\mathbf{dt}(t)$, and $\mathbf{sc}(t)$ implies $\mathbf{dt}(t)$. We may thus restrict our attention to types that satisfy the predicate $\mathbf{sc}$. For expressions this means that we do not need to interpret `Curry` (as $(t'\underline{\times}t"\underline{\rightarrow}t) \rightarrow (t'\underline{\rightarrow}t"\underline{\rightarrow}t)$ no longer is well-formed), `Apply` (as $(t\underline{\rightarrow}t'\underline{\rightarrow}t") \rightarrow (t\underline{\rightarrow}t') \rightarrow (t\underline{\rightarrow}t")$ no longer is well-formed), `Fix` (as $(t\underline{\rightarrow}t'\underline{\rightarrow}t') \rightarrow (t\underline{\rightarrow}t')$ no longer is well-formed) or `Const` (as $t'\rightarrow t\underline{\rightarrow}t'$ no longer is well-formed). The expression part $\mathbf{L}^{\mathrm{e}}$ of an interpretation for TML[$\mathbf{sc}$,$\mathbf{sc}$] may thus be specified by:

$\mathbf{L}^{\mathrm{e}}_{\mathrm{i}[t]}$ is some a priori fixed element of $[\![t]\!](\mathbf{L})$

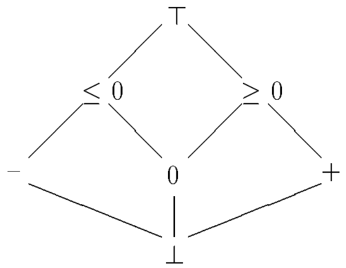
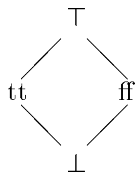
$$\text{e.g. } \mathbf{L}^{\mathrm{e}}_{=} = \lambda w.\ \begin{cases} (\top,\top) & \text{if } w = \mathit{live} \\ (\bot,\bot) & \text{if } w = \mathit{dead} \end{cases}$$

$\mathbf{L}^{\mathrm{e}}_{\mathrm{Tuple}} = \lambda v_1.\ \lambda v_2.\ \lambda(w_1,w_2).\ v_1(w_1) \sqcup v_2(w_2)$
$\mathbf{L}^{\mathrm{e}}_{\mathrm{Fst}} = \lambda v.\ \lambda w.\ v(w,\bot)$
$\mathbf{L}^{\mathrm{e}}_{\mathrm{Snd}} = \lambda v.\ \lambda w.\ v(\bot,w)$
$\mathbf{L}^{\mathrm{e}}_{\mathrm{Id}} = \lambda w.w$
$\mathbf{L}^{\mathrm{e}}_{\square} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_2(v_1(w))$
$\mathbf{L}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ v_1(\mathit{live}) \sqcup v_2(w) \sqcup v_3(w)$

Here we should note, in particular, that $\mathbf{L}^{\mathrm{e}}_{\square}$ uses the reverse order of composition wrt. $\mathbf{S}^{\mathrm{e}}_{\square}$ and $\mathbf{I}^{\mathrm{e}}_{\square}$.
— This ends Example 3.2.13. □

Example 3.2.14. (Detection of Signs — **II**) In our final example we shall consider an analysis where the type constructor $\underline{\times}$ should be interpreted in a different manner than in the standard semantics. The analysis we consider is once more the detection of signs analysis but this time using a relational method where the interdependence between components in a pair is taken into account. The formalisation amounts to defining a lattice interpretation $\mathbf{R}$ but a complete treatment requires a fair amount of machinery so we refer to [Nielson, 1984,Nielson, 1985b,Nielson, 1989] and only sketch the construction. For types we have:

- the property $\mathbf{R}^{\mathrm{t}}_{\mathrm{P}}$ equals 'is an algebraic lattice',
- the lattices $\mathbf{R}^{\mathrm{t}}_{\mathrm{i}}$ include

$\mathbf{R}^{\mathrm{t}}_{\mathrm{int}} =$  $\mathbf{R}^{\mathrm{t}}_{\mathrm{bool}} =$ 

- $\mathbf{R}^{\mathrm{t}}_{\underline{\times}} = \otimes$ (tensor product) and $\mathbf{R}^{\mathrm{t}}_{\underline{\to}} = \to$ (continuous function space).

Note here that the relational nature of the analysis is recorded by interpreting $\underline{\times}$ as a so-called tensor product, $\otimes$, just as the independent attribute nature of an analysis is recorded (in Example 3.2.11) by interpreting $\underline{\times}$ as a cartesian product, $\times$. We now need to define and motivate the tensor product $\otimes$.

Definition A tensor product of two algebraic lattices L_1 and L_2 is an algebraic lattice $L_1 \otimes L_2$ and a separately binary additive[4] and continuous function cross : $L_1 \times L_2 \to L_1 \otimes L_2$ that has the following universal property: Whenever $f : L_1 \times L_2 \to L$ is a separately binary additive and continous function between algebraic lattices then there exists precisely one continuous and binary additive function $f^{\times} : L_1 \otimes L_2 \to L$ (called the extension of f) such that

$$\begin{tikzcd} L_1 \times L_2 \arrow[d, "\text{cross}"] \arrow[dr, "f"] & \\ L_1 \otimes L_2 \arrow[r, "f^{\times}"'] & L \end{tikzcd}$$

commutes, i.e. such that $f^{\times} \circ \text{cross} = f$.

Proposition A tensor product always exists (and it is unique to within isomorphism).

Proof: See [Bandelt, 1980] (or [Nielson, 1984] for an elementary proof). It is important for this result that some lattice structure is assumed as the tensor product does not exist for arbitrary domains. $\Box$

Having been assured of the existence of the tensor product the next task is to motivate why it is relevant. We do so by calculating the tensor product in a special case and by showing that the tensor product has the ability to

[4]A function $f : L_1 \times L_2 \to L$ is separately binary additive if $\lambda l_1.f(l_1, l_2')$ and $\lambda l_2.f(l_1', l_2)$ are binary additive for all $l_1' \in L_1$ and $l_2' \in L_2$.

express the interdependence between components in a pair. For this let S be a set and $S_\perp$ the domain with elements $S\cup\{\perp\}$ and the partial order $\sqsubseteq$ given by $s_1\sqsubseteq s_2$ iff $s_1=s_2$ or $s_1=\perp$. For a domain D in which all elements are compact, as holds for $S_\perp$ and $S_\perp\times S_\perp$, the lower powerdomain $\mathcal{P}_l(D)$ may be defined as

$$(\ \{Y\subseteq D \mid \perp\in Y \wedge \forall d\in D\colon \forall y\in Y\colon d\sqsubseteq y \Rightarrow d\in Y\},\ \subseteq)$$

This is an algebraic lattice and $\mathcal{P}_l(S_\perp)$ is isomorphic to the powerset $\mathcal{P}(S)$. One can verify that setting $\mathcal{P}_l(S_\perp)\otimes\mathcal{P}_l(S_\perp) = \mathcal{P}_l(S_\perp\times S_\perp)$ and cross $=$ $\lambda(Y_1,Y_2).\,Y_1\times Y_2$ satisfies the definition of a tensor product and that the extension of a function f is given by

$$f^\times = \lambda Y.\ \bigsqcup\{\ f(\{d_1|d_1\sqsubseteq y_1\},\ \{d_2|d_2\sqsubseteq y_2\}) \mid (y_1,y_2)\in Y\ \}$$

This shows that in the particular case of a tensor product of powerdomains, the tensor product has the ability to express the interdependence between components in a pair.

Remark Another kind of motivation amounts to explaining the role of binary additive functions. In general an algebraic lattice imposes certain limitations upon the combinations of properties that can be expressed, for example that one cannot express the property 'l_1 and l_2 but not l'. (In the lattice $\mathbf{R}^{\mathrm{t}}_{\mathrm{int}}$ for the detection of signs one may take l_1=–, l=0 and l_2=+.) The binary additive functions are those functions that somehow respect these limitations. The constraining factor in the definition of $L_1\otimes L_2$ then is that when considering each component (as is evidenced by the demands on f) one should respect the limitations inherent in the L_i. This means that e.g. cross(+,–)$\sqcup$cross(–,–) must also describe cross(0,–).

We have to refer to [Nielson, 1984,Nielson, 1985b] for a further discussion of the role of tensor products. For completeness we shall also finish by sketching the expression part of the lattice interpretation $\mathbf{R}$:

$\mathbf{R}^{\mathrm{e}}_{\mathrm{i}[t]}$ is some a priori fixed element of $[\![t]\!](\mathbf{R})$
e.g. $\mathbf{R}^{\mathrm{e}}_{\mathrm{true}} = \mathrm{tt}$, $\mathbf{R}^{\mathrm{e}}_{1} = +$ etc.
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Tuple}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ \mathrm{cross}(v_1(w),v_2(w))$
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Fst}} = \lambda v.\ \lambda w.\ d_1$ where $(d_1,d_2) = \mathrm{id}^\times(v(w))$
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Snd}} = \lambda v.\ \lambda w.\ d_2$ where $(d_1,d_2) = \mathrm{id}^\times(v(w))$
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Curry}} = \lambda v.\ \lambda w_1.\ \lambda w_2.\ v(\mathrm{cross}(w_1,w_2))$
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Apply}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(w)(v_2(w))$
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Id}} = \lambda w.w$
$\mathbf{R}^{\mathrm{e}}_{\Box} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$
$\mathbf{R}^{\mathrm{e}}_{\mathrm{Const}} = \lambda v.\ \lambda w.\ v$

$$\mathbf{R}^{\mathrm{e}}_{\mathrm{Fix}} = \lambda v.\ \lambda w.\ \mathrm{FIX}(v(w))$$

$$\mathbf{R}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ \begin{cases} v_2(w) & \text{if } v_1(w)=\mathrm{tt} \\ v_3(w) & \text{if } v_1(w)=\mathrm{ff} \\ \bot & \text{if } v_1(w)=\bot \\ v_2(w) \sqcup v_3(w) & \text{if } v_1(w)=\top \end{cases}$$

However, we should point out that with respect to the treatment given in [Nielson, 1984,Nielson, 1989] the equations for `Tuple` and `If` are correct but too imprecise. To improve this we would need to break an argument into its *atoms* (these are the elements immediately above $\bot$) and process these separately.
— This ends Example 3.2.14. □

3.2.4 Summary

To summarise, we have seen that for the purposes of abstract interpretation we need the ability to interpret underlined base types in different ways in order to describe the properties used in the different analyses. Furthermore, we have seen that the well-known distinction between forward and backward analyses may be formalized by the way $\underline{\rightarrow}$ is interpreted (!) and that the well-known distinction between independent attribute and relational methods may be formalized by the way $\underline{\times}$ is interpreted (!). This gives credit to the claim that a two-level metalanguage is a natural setting in which to develop a theory of abstract interpretation.

3.3 Correctness of Analyses

To have faith in an analysis one must be able to prove that the properties resulting from the analysis are correct, e.g. with respect to the values that the standard semantics operates on. First of all this necessitates a framework in which one can *formulate* the desired correctness relations. Secondly it is desirable that the correctness follows for all terms in the metalanguage (hence all programs considered in Figure 1) once the correctness of the basic expressions and combinators has been established. (In the terminology of Subsection 2.8 this amounts to showing that local correctness is a sufficient condition for global correctness.) Then one can consider the analyses one by one and complete the definition of the correctness relations and use this to prove the correctness of the basic expressions and combinators.

To *formulate* the correctness relations we shall adopt the framework of logical relations [Plotkin, 1980] (essentially called relational functors in [Reynolds, 1974]). For this we shall assume the existence of two domain or lattice interpretations $\mathcal{I}$ and $\mathcal{J}$ and the task is to define an admissible relation $\mathcal{R}[\![t]\!]$ between $[\![t]\!](\mathcal{I})$ and $[\![t]\!](\mathcal{J})$, i.e.

$$\mathcal{R}[\![t]\!] : [\![t]\!](\mathcal{I}) \times [\![t]\!](\mathcal{J}) \rightarrow \{\text{true,false}\}$$

in such a way that $\mathcal{R}[\![t]\!]$ formalizes our intuitions about correctness. We shall feel free to write $d\ \mathcal{R}[\![t]\!]\ e$ as well as $\mathcal{R}[\![t]\!](d,e)$. In the interest of readability we prefer the notation $\mathcal{R}[\![t]\!]$ for $[\![t]\!](\mathcal{R})$ but regardless of this $\mathcal{R}$ should be considered a parameter to $[\![t]\!](\cdots)$. The definition of $\mathcal{R}[\![t]\!]$ is by induction on the structure of t:

$$\begin{aligned}
&\mathcal{R}[\![\mathtt{A}_{\mathrm{i}}]\!](d,e) \equiv d{=}e\\
&\mathcal{R}[\![t_1{\times}t_2]\!]((d_1,d_2),(e_1,e_2)) \equiv \mathcal{R}[\![t_1]\!](d_1,e_1) \wedge \mathcal{R}[\![t_2]\!](d_2,e_2)\\
&\mathcal{R}[\![t_1{\rightarrow}t_2]\!](f,g) \equiv \forall d,e\colon \mathcal{R}[\![t_1]\!](d,e) \Rightarrow \mathcal{R}[\![t_2]\!](f(d),g(e))\\
&\mathcal{R}[\![\underline{\mathtt{A}}_{\mathrm{i}}]\!] \equiv \mathcal{R}_{\mathrm{i}}\\
&\mathcal{R}[\![t_1\underline{\times}t_2]\!] \equiv \mathcal{R}_{\times}(\mathcal{R}[\![t_1]\!],\mathcal{R}[\![t_2]\!])\\
&\mathcal{R}[\![t_1\underline{\rightarrow}t_2]\!](f,g) \equiv \mathcal{R}_{\rightarrow}(\mathcal{R}[\![t_1]\!],\mathcal{R}[\![t_2]\!])
\end{aligned}$$

The demands on $\mathcal{R}$, i.e. $(\mathcal{R}_{\mathrm{i}})_{\mathrm{i}}$, $\mathcal{R}_{\times}$ and $\mathcal{R}_{\rightarrow}$, are made clear in:

Definition 3.3.1. A *correctness correspondence* (or just *correspondence*) $\mathcal{R}$ between domain or lattice interpretations $\mathcal{I}$ and $\mathcal{J}$ is a specification of

- admissible relations $\mathcal{R}_{\mathrm{i}} : \mathcal{I}_{\mathrm{i}}^{\mathrm{t}} \times \mathcal{J}_{\mathrm{i}}^{\mathrm{t}} \rightarrow \{\text{true,false}\}$,
- operations $\mathcal{R}_{\times}$ and $\mathcal{R}_{\rightarrow}$ upon admissible relations such that

  $$\begin{aligned}
  &\mathcal{R}_{\times}(\mathsf{R}_1,\mathsf{R}_2) : \mathcal{I}_{\times}^{\mathrm{t}}(D_1,D_2) \times \mathcal{J}_{\times}^{\mathrm{t}}(E_1,E_2) \rightarrow \{\text{true,false}\}\\
  &\mathcal{R}_{\rightarrow}(\mathsf{R}_1,\mathsf{R}_2) : \mathcal{I}_{\rightarrow}^{\mathrm{t}}(D_1,D_2) \times \mathcal{J}_{\rightarrow}^{\mathrm{t}}(E_1,E_2) \rightarrow \{\text{true,false}\}
  \end{aligned}$$

  are admissible relations whenever $\mathsf{R}_{\mathrm{i}}{:}D_{\mathrm{i}}{\times}E_{\mathrm{i}}{\rightarrow}\{\text{true,false}\}$ are admissible relations, D_1 and D_2 are domains that satisfy the property $\mathcal{I}_{\mathrm{P}}^{\mathrm{t}}$ and E_1 and E_2 are domains that satisfy the property $\mathcal{J}_{\mathrm{P}}^{\mathrm{t}}$.

Proposition 3.3.2. The equations for $\mathcal{R}[\![t]\!]$ define an admissible relation when t is a well-formed type in TML[**dt**,**dt**] and $\mathcal{R}$ is a correctness correspondence as above. □

Proof: We must show by structural induction on t that $\mathcal{R}[\![t]\!]$ is an admissible relation between $[\![t]\!](\mathcal{I})$ and $[\![t]\!](\mathcal{J})$ and that these domains exist. This is a straightforward structural induction and we omit the details. (Note that the only reason for demanding t to be a well-formed type in TML[**dt**,**dt**], i.e. $\vdash_{dt}\ t$, is for $[\![t]\!](\mathcal{I})$ and $[\![t]\!](\mathcal{J})$ to be guaranteed to exist). □

The *correctness of the basic expressions and combinators* amounts to showing that the appropriate correctness relations hold between their interpretations as given by $\mathcal{I}$ and $\mathcal{J}$. We shall write $\mathcal{R}(\mathcal{I},\mathcal{J})$, or $\mathcal{I}\ \mathcal{R}\ \mathcal{J}$, for this and the formal definition is:

> whenever ψ is a basic expression or combinator and the following hold

$$\mathcal{I}^{e}_{\psi} \in [\![t_1]\!](\mathcal{I}) \to \cdots [\![t_n]\!](\mathcal{I}) \to [\![t]\!](\mathcal{I})$$
$$\mathcal{J}^{e}_{\psi} \in [\![t_1]\!](\mathcal{J}) \to \cdots [\![t_n]\!](\mathcal{J}) \to [\![t]\!](\mathcal{J})$$

for well-formed types t_1, ..., t_n and t in TML[**dt**,**dt**], then we have

$$\mathcal{R}[\![t_1]\!](d_1,e_1) \wedge \cdots \wedge \mathcal{R}[\![t_n]\!](d_n,e_n)$$
$$\Rightarrow \mathcal{R}[\![t]\!](\, \mathcal{I}^{e}_{\psi}(d_1)\cdots(d_n),\, \mathcal{J}^{e}_{\psi}(e_1)\cdots(e_n))$$

A shorter statement of the desired relation between $\mathcal{I}^{e}_{\psi}$ and $\mathcal{J}^{e}_{\psi}$ is that $\mathcal{R}[\![t_1 \to \cdots t_n \to t]\!](\mathcal{I}^{e}_{\psi},\ \mathcal{J}^{e}_{\psi})$ must hold. This exploits the fact that $t_1 \to \cdots t_n \to t$ is well-formed (i.e. satisfies **dt**) if and only if all of t_1, ..., t_n and t are and we may thus regard $\mathcal{I}^{e}_{\psi}$ as an element of $[\![t_1 \to \cdots t_n \to t]\!](\mathcal{I})$ and similarly for $\mathcal{J}^{e}_{\psi}$.

It now follows that the correctness of an analysis amounts to the correctness of the basic expressions and combinators:

Proposition 3.3.3. To show the correctness $\mathcal{R}([\![e]\!](\mathcal{I}),[\![e]\!](\mathcal{J}))$ of a closed expression e in TML[**dt**,**dt**] it suffices to prove R($\mathcal{I}$,$\mathcal{J}$). □

Proof: Let $\mathcal{R}$ be a correctness correspondence between the domain or lattice interpretations $\mathcal{I}$ and $\mathcal{J}$ and such that $\mathcal{I}\ \mathcal{R}\ \mathcal{J}$ holds. We then prove by structural induction on a well-formed expression e that

if $tenv \vdash_{dt,dt} \ e : t$ with dom($tenv$)=$\{\mathtt{x}_1,\cdots,\mathtt{x}_n\}$, $tenv(\mathtt{x}_i)=t_i$
and $\vdash_{dt}\ t_i$
then $\mathcal{R}[\![t_1]\!](d_1,e_1) \wedge \cdots \wedge \mathcal{R}[\![t_n]\!](d_n,e_n) \Rightarrow \mathcal{R}[\![t]\!](\,[\![e]\!](\mathcal{I})(d_1,\cdots,d_n),$
$[\![e]\!](\mathcal{J})(e_1,\cdots,e_n))$

The structural induction is mostly straightforward. In the case where e = `fix` e_0 we use the induction principle of Definition 3.2.3. □

Example 3.3.4. We shall now use the above development to show the correctness of the detection of signs analysis of Example 3.2.11 with respect to the lazy standard semantics of Example 3.2.10. The first task is to define a correctness correspondence cor between the domain interpretation **S** and the lattice interpretation **I**. We have

- the admissible relations cor_i include

$$\mathsf{cor}_{\mathrm{int}}(d,p) \equiv p \sqsupseteq \begin{cases} - & \text{if } d<0 \wedge d\neq\bot \\ 0 & \text{if } d=0 \\ + & \text{if } d>0 \wedge d\neq\bot \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{cor}_{\mathrm{bool}}(d,p) \equiv p \sqsupseteq \begin{cases} \mathrm{tt} & \text{if } d\text{=true} \\ \mathrm{ff} & \text{if } d\text{=false} \\ \bot & \text{otherwise} \end{cases}$$

so that e.g. $\mathsf{cor}_{\mathrm{int}}(7,\neg\!-)$ and $\mathsf{cor}_{\mathrm{bool}}(\mathrm{true},\top)$,

- the operations upon admissible relations are

$$\mathsf{cor}_{\times}(\mathsf{R}_1,\mathsf{R}_2)\ ((d_1,d_2),(p_1,p_2)) \equiv \mathsf{R}_1(d_1,p_1) \wedge \mathsf{R}_2(d_2,p_2)$$
$$\mathsf{cor}_{\rightarrow}(\mathsf{R}_1,\mathsf{R}_2)\ (f,h) \equiv \forall d,p\text{: } \mathsf{R}_1(d,p) \Rightarrow \mathsf{R}_2(f(d),h(p))$$

(much as for $[\![\cdots\times\cdots]\!](\mathcal{R})$ and $[\![\cdots\rightarrow\cdots]\!](\mathcal{R})$ above),

and it is straightforward to verify that this does specify a correctness correspondence.

The next task is to show $\mathsf{cor}(\mathbf{S},\mathbf{I})$ so that Proposition 3.3.3 can be invoked. For the basic expressions $\mathtt{f}_{\mathrm{i}}[t]$ we must show

$$\mathsf{cor}[\![t]\!](\mathbf{S}_{\mathrm{i}}^{\mathrm{e}},\mathbf{I}_{\mathrm{i}}^{\mathrm{e}})$$

Little can be said here as we have not mentioned many $f_{\mathrm{i}}[t]$ in Examples 3.2.10 and 3.2.11 but we may note that

$$\mathsf{cor}[\![\underline{\mathtt{Bool}}]\!](\mathrm{true},\mathrm{tt})$$
$$\mathsf{cor}[\![\underline{\mathtt{Int}}]\!](1,+)$$

both hold. For the combinators `Tuple`, `Fst` and `Snd` related to product it is straightforward to verify the required relations as the definition of these combinators is 'the same' in **S** and **I**. A similar remark holds for the combinators `Curry`, `Apply`, `Id`, $\Box$ and `Const` related to function space. For the combinator `Fix` we may assume

$$\mathsf{R}(ws_1,wi_1) \wedge \mathsf{R}'(ws_2,wi_2) \Rightarrow \mathsf{R}'(vs(ws_1)(ws_2),vi(wi_1)(wi_2))$$
$$\mathsf{R}'(ws,wi)$$

and must show

$$\mathsf{R}'(\mathrm{FIX}(vs(ws)),\mathrm{FIX}(vi(wi)))$$

where R is $\mathsf{cor}[\![t]\!]$ and R' is $\mathsf{cor}[\![t']\!]$ for types t and t' that both satisfy the predicate **lt**. As in the proof of Proposition 3.3.3 this follows using the induction principle of Definition 3.2.3. Finally for the combinator `If` we may assume

$$\mathsf{R}(ws,wi) \Rightarrow vi_1(wi) \sqsupseteq \begin{cases} \mathrm{tt} & \text{if } vs_1(ws)\text{=true} \\ \mathrm{ff} & \text{if } vs_1(ws)\text{=false} \\ \bot & \text{otherwise} \end{cases}$$
$$\mathsf{R}(ws,wi) \Rightarrow \mathsf{R}'(vs_2(ws),vi_2(wi))$$
$$\mathsf{R}(ws,wi) \Rightarrow \mathsf{R}'(vs_3(ws),vi_3(wi))$$
$$\mathsf{R}(ws,wi)$$

and must show

$$\mathsf{R}'\left(\left\{\begin{array}{ll} vs_2(ws) & \text{if } vs_1(ws)\text{=true} \\ vs_3(ws) & \text{if } vs_1(ws)\text{=false} \\ \bot & \text{otherwise} \end{array}\right\}, \left\{\begin{array}{ll} vi_2(wi) & \text{if } vi_1(wi)\text{=tt} \\ vi_3(wi) & \text{if } vi_1(wi)\text{=ff} \\ vi_2(wi)\sqcup vi_3(wi) & \text{if } vi_1(wi)\text{=}\top \\ \bot & \text{otherwise} \end{array}\right\}\right)$$

where again R is $\mathsf{cor}[\![t]\!]$ and R' is $\mathsf{cor}[\![t']\!]$ for types t and t' that both satisfy the predicate **lt**. The proof amounts to considering each of the cases $vs_1(ws)$=true, $vs_1(ws)$=false and $vs_1(ws)=\bot$ separately and will need:

Fact If t satisfies **lt** then $\mathsf{cor}[\![t]\!](ws,wi) \wedge wi \sqsubseteq wi'$ implies $\mathsf{cor}[\![t]\!](ws,wi')$.

The proof of this fact is by structural induction on t. The case $t = \underline{\mathtt{A}}_{\mathrm{i}}$ can only be conducted if we tacitly assume that $\underline{\mathtt{A}}_{\mathrm{i}}$ is one of $\underline{\mathtt{A}}_{\mathrm{bool}}$ or $\underline{\mathtt{A}}_{\mathrm{int}}$. □

For reasons of space we shall not prove the correctness of the remaining analyses defined in Subsection 3.2. There are no profound difficulties in establishing the correctness of the detection of signs analysis defined in Example 3.2.14. For the liveness analysis of Example 3.2.13 the notion of a correctness correspondence is too weak but a variation of the development presented here may be used to prove its correctness (see [Nielson, 1989]). We should point out that the complications in the proof of correctness of the liveness analysis are due to the fact that the properties in the liveness analysis do not describe actual values but rather their subsequent use in future computations. The terms *first-order* analyses (e.g. detection of signs) and *second-order* analyses (e.g. liveness) have been used for this distinction [Nielson, 1985a,Nielson, 1989].

3.3.1 Safety: Comparing two analyses

A special case of correctness is when one compares two analyses and shows that the properties resulting from one analysis correctly describe the properties resulting from the other. We shall use the term *safety* for this and we shall see in the next subsection that from the safety of one analysis with respect to a correct analysis one is often able to infer the correctness of the former analysis.

As an example we might consider two analyses that operate on the same properties but have different ways of modelling the basic expressions and combinators. We formalize this by considering two lattice interpretations $\mathcal{I}$ and $\mathcal{J}$ with $\mathcal{I}^{\mathrm{t}}_{\mathrm{P}}=\mathcal{J}^{\mathrm{t}}_{\mathrm{P}}$, $\mathcal{I}^{\mathrm{t}}_{\mathrm{i}}=\mathcal{J}^{\mathrm{t}}_{\mathrm{i}}$, $\mathcal{I}^{\mathrm{t}}_{\times}=\mathcal{J}^{\mathrm{t}}_{\times}$ and $\mathcal{I}^{\mathrm{t}}_{\rightarrow}=\mathcal{J}^{\mathrm{t}}_{\rightarrow}$ (in short $\mathcal{I}^{\mathrm{t}}=\mathcal{J}^{\mathrm{t}}$). When we want to be more specific we shall let $\mathcal{I}$ be the interpretation **I** for the detection of signs (Example 3.2.10). If we wish to express that the results of $\mathcal{J}$ are coarser than those of $\mathcal{I}$, e.g. that $[\![e]\!](\mathcal{I})$=+ whereas $[\![e]\!](\mathcal{J})=\neg -$, we must define a correctness correspondence and we shall use the notation $\leq$. Given the motivation presented in Section 2 we take

$$\leq_{\mathrm{i}} \equiv \sqsubseteq$$

$\leq_{\times}(\mathsf{R}_1,\mathsf{R}_2) \equiv \sqsubseteq$
$\leq_{\rightarrow}(\mathsf{R}_1,\mathsf{R}_2) \equiv \sqsubseteq$

because the idea was to use the partial order $\sqsubseteq$ to express the amount of precision among various properties[5]. In a more abstract way one might say that a correspondence $\mathcal{R}$ between two lattice interpretations $\mathcal{I}$ and $\mathcal{J}$ is a *safety correspondence* when $\mathcal{R}_{\mathrm{i}}{\equiv}\sqsubseteq$, $\mathcal{R}_{\times}(\sqsubseteq,\sqsubseteq){\equiv}\sqsubseteq$ and $\mathcal{R}_{\rightarrow}(\sqsubseteq,\sqsubseteq){\equiv}\sqsubseteq$ whenever $\mathcal{I}^{\mathrm{t}}{=}\mathcal{J}^{\mathrm{t}}$. Clearly $\leq$ is a safety correspondence.

We shall claim that $\leq$ is the proper relation to use for relating $\mathcal{I}$ and $\mathcal{J}$. On an all-underlined type t (e.g. $\underline{\texttt{Int}}$ or $\underline{\texttt{Int}{\rightarrow}\texttt{Int}}$) the relation $\leq[\![t]\!]$ clearly equals $\sqsubseteq$ which is the relation that also Section 2 used. On a type t without any underlined symbols (e.g. `Int` or `Int→Int`) it is straightforward to see (as we show below) that $\leq[\![t]\!]$ equals $=$ and this is the correct relation to use given that we only perform abstract interpretation on underlined base types and constructors. In particular, $\leq[\![t]\!]$ is more adequate than $\sqsubseteq$ in this case.

However, there are types upon which $\leq[\![t]\!]$ behaves in a strange way. As an example let $t_0{=}\underline{\texttt{Bool}}{\rightarrow}\texttt{Bool}$ and consider a basic expression $\mathtt{f}_0[t_0]$ such that

$$\mathcal{I}_0^{\mathrm{e}} = \mathcal{J}_0^{\mathrm{e}} = \lambda d.\begin{cases} \text{true} & \text{if } d{=}\top \\ \bot & \text{otherwise} \end{cases}$$

Then $\mathcal{I}_0^{\mathrm{e}} \leq[\![t_0]\!]\ \mathcal{J}_0^{\mathrm{e}}$ *fails* because we have $\bot{\sqsubseteq}\top$ but not $\bot{=}$true. This means that $\leq[\![t_0]\!]$ is not even reflexive. Clearly we want $\leq[\![t]\!]$ to be a partial order and it is also natural to assume that it implies $\sqsubseteq$ because we have argued for the use of $\sqsubseteq$ to compare properties of an all-underlined type. We shall achieve this by restricting the types to be considered just as we did to ensure that $[\![t]\!](\mathcal{I})$ and $[\![t]\!](\mathcal{J})$ were domains.

First we need a few definitions:

Definition 3.3.5. A *suborder* $\leq$ on a domain $D{=}(D,\sqsubseteq)$ is a partial order that satisfies

$$d_1{\leq}d_2 \Rightarrow d_1{\sqsubseteq}d_2$$

for all d_1 and d_2. □

For an arbitrary safety correspondence $\mathcal{R}$, e.g. $\leq$, this motivates defining the predicates

pt(t) to ensure that $\mathcal{R}[\![t]\!]$ amounts to $=$,

it(t) to ensure that $\mathcal{R}[\![t]\!]$ amounts to $\sqsubseteq$,

[5] This need not be so in general (see [Mycroft, 1983]) but considerably simplifies the technical development.

$\mathbf{lpt}(t)$ to ensure that $\mathcal{R}[\![t]\!]$ is a suborder.

The predicate **pt** was called *pure* in [Nielson, 1988a,Nielson, 1989] because it will restrict the types to have no underlined symbols. The predicates **it** and **lpt** should be thought of as slightly more discriminating analogues of **lt** and **dt**. They were called *impure* and *level-preserving*, respectively, in [Nielson, 1988a,Nielson, 1989] but with one difference: TML[**dt**,**dt**] allows more well-formed types than does [Nielson, 1988a] or [Nielson, 1989].

Definition 3.3.6. The predicates **pt** (for *pure type*), **it** (for *impure type*) and **lpt** (for *level-preserving type*) are defined by:

	$\mathtt{A_i}$	$\underline{\mathtt{A_i}}$	$t_1 \times t_2$	$t_1 \underline{\times} t_2$	$t_1 \to t_2$	$t_1 \underline{\to} t_2$
pt	true	false	$\mathbf{pt}_1 \wedge \mathbf{pt}_2$	false	$\mathbf{pt}_1 \wedge \mathbf{pt}_2$	false
it	false	true	$\mathbf{it}_1 \wedge \mathbf{it}_2$	$\mathbf{it}_1 \wedge \mathbf{it}_2$	$\mathbf{lpt}_1 \wedge \mathbf{it}_2$	$\mathbf{it}_1 \wedge \mathbf{it}_2$
lpt	true	true	$\mathbf{lpt}_1 \wedge \mathbf{lpt}_2$	$\mathbf{it}_1 \wedge \mathbf{it}_2$	$(\mathbf{pt}_1 \wedge \mathbf{lpt}_2) \vee (\mathbf{lpt}_1 \wedge \mathbf{it}_2)$	$\mathbf{it}_1 \wedge \mathbf{it}_2$

Note that the difference between **lt** and **dt** versus **it** and **lpt** is due to the difference between the definition of $\mathbf{dt}(t_1 \to t_2)$ and $\mathbf{lpt}(t_1 \to t_2)$.

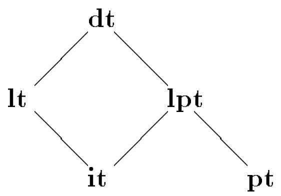


Fig. 2. Wellformedness-constraints

Lemma 3.3.7. The formal definition of the predicates **pt**, **it** and **lpt** satisfy the intentions displayed above. $\square$

Proof: For an arbitrary safety relation $\mathcal{R}$, e.g. $\leq$, between lattice interpretations $\mathcal{I}$ and $\mathcal{J}$ with $\mathcal{I}^{\mathrm{t}} = \mathcal{J}^{\mathrm{t}}$ we prove by induction on types t that

- $\mathbf{pt}(t) \Rightarrow \mathbf{lpt}(t) \wedge \mathcal{R}[\![t]\!] \equiv =$,
- $\mathbf{it}(t) \Rightarrow \mathbf{lpt}(t) \wedge \mathbf{lt}(t) \wedge \mathcal{R}[\![t]\!] \equiv \sqsubseteq$,
- $\mathbf{lpt}(t) \Rightarrow \mathbf{dt}(t) \wedge \mathcal{R}[\![t]\!]$ is a suborder.

The first result is a straightforward structural induction and we shall not give any details. The second and third result must be proved jointly as **it** and **lpt** are mutually interdependent.

The cases $\mathtt{A}_i$ and $\underline{\mathtt{A}}_i$ are straightforward. In the case $t{=}t_1{\times}t_2$ we first assume that $\mathbf{it}(t)$. Then $\mathbf{it}(t_1)$ and $\mathbf{it}(t_2)$ so that $\mathbf{lpt}(t_1)$, $\mathbf{lpt}(t_2)$, $\mathbf{lt}(t_1)$, $\mathbf{lt}(t_2)$, $\mathcal{R}[\![t_1]\!]{\equiv}{\sqsubseteq}$ and $\mathcal{R}[\![t_2]\!]{\equiv}{\sqsubseteq}$. It follows that $\mathbf{lpt}(t)$, $\mathbf{lt}(t)$ and $\mathcal{R}[\![t]\!]{\equiv}{\sqsubseteq}$. Next we assume that $\mathbf{lpt}(t)$. Then $\mathbf{lpt}(t_1)$ and $\mathbf{lpt}(t_2)$ so that $\mathbf{dt}(t_1)$, $\mathbf{dt}(t_2)$, $\mathcal{R}[\![t_1]\!]$ is a suborder and $\mathcal{R}[\![t_2]\!]$ is a suborder. It follows that $\mathbf{dt}(t)$ and that $\mathcal{R}[\![t]\!]$ is a suborder. In the case $t{=}t_1\underline{\times}t_2$ the assumptions $\mathbf{it}(t)$ and $\mathbf{lpt}(t)$ are equivalent so we assume that $\mathbf{it}(t)$ holds. Then $\mathbf{it}(t_1)$ and $\mathbf{it}(t_2)$ so that $\mathbf{lt}(t_1)$, $\mathbf{lt}(t_2)$, $\mathcal{R}[\![t_1]\!]{\equiv}{\sqsubseteq}$ and $\mathcal{R}[\![t_2]\!]{\equiv}{\sqsubseteq}$. It follows that $\mathbf{lpt}(t)$, $\mathbf{lt}(t)$, $\mathbf{dt}(t)$ and $\mathcal{R}[\![t]\!]{\equiv}{\sqsubseteq}$ which is a suborder. The case $t{=}t_1\underline{\rightarrow}t_2$ is similar.

In the final case $t{=}t_1{\rightarrow}t_2$ we first assume $\mathbf{it}(t)$. Then $\mathbf{lpt}(t_1)$ and $\mathbf{it}(t_2)$ so that $\mathbf{dt}(t_1)$, $\mathbf{lt}(t_2)$, $\mathcal{R}[\![t_1]\!]$ is a suborder and $\mathcal{R}[\![t_2]\!]{\equiv}{\sqsubseteq}$. It follows that $\mathbf{lpt}(t)$, $\mathbf{lt}(t)$ and hence also $\mathbf{dt}(t)$. The relation $\mathcal{R}[\![t]\!]$ holds on (f,g) when

$$\mathcal{R}[\![t_1]\!](d,e) \Rightarrow f(d){\sqsubseteq}g(e)$$

for all d and e. If $f{\sqsubseteq}g$ and $\mathcal{R}[\![t_1]\!](d,e)$ we have $d{\sqsubseteq}e$ and hence $f(d){\sqsubseteq}g(e)$ so that $\mathcal{R}[\![t]\!](f,g)$. If $\mathcal{R}[\![t]\!](f,g)$ we have $\mathcal{R}[\![t_1]\!](d,d)$ and hence $f(d){\sqsubseteq}g(d)$ for all d and this amounts to $f{\sqsubseteq}g$. Thus $\mathcal{R}[\![t]\!]$ equals the suborder $\sqsubseteq$. Next we assume that $\mathbf{lpt}(t)$. There are two cases to consider but we have just treated $\mathbf{lpt}(t_1){\wedge}\mathbf{it}(t_2)$ so that we may assume $\mathbf{pt}(t_1)$ and $\mathbf{lpt}(t_2)$. It follows that $\mathbf{lpt}(t_1)$, $\mathbf{dt}(t_1)$, $\mathcal{R}[\![t_1]\!]{\equiv}{=}$, $\mathbf{dt}(t_2)$ and $\mathcal{R}[\![t_2]\!]$ is a suborder. Hence $\mathbf{dt}(t)$ and the relation $\mathcal{R}[\![t]\!]$ holds on (f,g) whenever

$$\forall d\colon\ \mathcal{R}[\![t_2]\!](f(d),g(d))$$

and this is clearly a suborder. — This ends the proof of Lemma 3.3.7. □

3.3.2 Summary

We shall now restrict the types of the basic expressions and combinators so that they have level-preserving types. This amounts to considering TML[**lpt**,**dt**] as there is no need also to require the types of variables to be level-preserving. For the basic expressions $\mathtt{f}_i[t]$ this condition simply amounts to requiring t to be level-preserving, i.e. satisfy the predicate **lpt**. For the combinators a general form of their types may be found in the sideconditions in Tables 1 and 2. These general forms are expressed in terms of subtypes t, t', t", t_0, t_1 and t_2 and the restriction to TML[**lpt**,**dt**] amounts to demanding that all these subtypes are impure, i.e. satisfy the predicate **it**.

The relationship between TML[**lpt**,**dt**] and TML[**dt**,**dt**] is clarified by:

Fact 3.3.8. If $tenv \vdash_{pt,dt}\ e : t$ then $tenv \vdash_{dt,dt}\ e : t$ and hence $\vdash_{dt}\ t$. □

Thus TML[**lpt**,**dt**] is a proper subset of TML[**dt**,**dt**] and Fact 3.1.4, Proposition 3.2.6, Proposition 3.2.9, Proposition 3.3.2 and Proposition 3.3.3 apply to TML[**lpt**,**dt**] as well.

Given Lemma 3.3.7 we then have that $\leq$ is a partial order in the collection of interpretations for TML[**lpt**,**dt**] contrary to what is the case when one considers the collection of all interpretations for TML[**dt**,**dt**]. In particular we have $\mathcal{I}\leq\mathcal{I}$ whenever $\mathcal{I}$ is an interpretation for TML[**lpt**,**dt**] and by Proposition 3.3.3 we then have $(\llbracket e\rrbracket(\mathcal{I})) \leq\llbracket t\rrbracket\ (\llbracket e\rrbracket(\mathcal{I}))$ for all closed expressions e of type t (even if t is not level-preserving).

3.4 Induced Analyses

One shortcoming of the development of the previous subsection is that a correct analysis may be so imprecise as to be practically useless. An example is an analysis where all basic expressions and combinators are interpreted as the greatest element $\top$ (whenever they are used with a lattice type). The notion of correctness is topological in nature but we would ideally like something that was a bit more metric in nature so that we could express *how* imprecise a correct analysis is. Unfortunately no one has been able to develop an adequate metric for these purposes.

The alternative then is to compare various analyses. We shall take the point of view that the choice of the type part of an interpretation, i.e. the choice of what properties to use for underlined types etc., represents a deliberate choice as to the degree of precision that is desired[6]. Thus the definition of $\leq$ in the previous subsection allows us to compare various analyses provided that they use the same selection of properties. So if we are confronted with two analyses we may compare them and might be able to say that one analysis is more imprecise than (i.e. $\geq$) another and so we might prefer the other analysis. This is not a complete recipe as $\leq$ is only a partial order and in general not a total order. Also even if we have preferred some analysis there is no easy way to tell whether we could develop an analysis that would be even more precise.

This motivates the development in the present subsection where we show that under certain circumstances there is a *most precise* analysis over a given selection of properties. Following [Cousot, 1979] we shall term this the *induced* analysis. As we shall see in the next subsection there may well be pragmatic reasons for adopting an analysis that is less precise than the *induced* analysis. However, even if one does so we believe that the induced analysis serves an important role as a standard against which analyses may be compared: whenever the analysis of one's choice models

[6] This is a more restricted point of view than is put forward in [Steffen, 1987].

a basic expression or combinator less precisely than the induced analysis does then one may judge the degree of imprecision and decide whether it is warranted for pragmatic reasons (e.g. termination, low time-complexity, easy to implement, etc.).

For the technical development we shall assume that we have a domain or lattice interpretation $\mathcal{I}$ and a lattice interpretation $\mathcal{J}$. (Actually, we only need the type part of $\mathcal{J}$ and for the majority of the development we also only need the type part of $\mathcal{I}$.) Here one should think of $\mathcal{I}$ as the standard semantics, e.g. the lazy standard semantics of Example 3.2.10, and one should think of $\mathcal{J}$ as some analysis, e.g. the detection of signs of Example 3.2.11. However, the development also specialises to the case where $\mathcal{I}$ is some analysis much as the notion of correctness correspondence in the previous subsection specialised to the notion of safety correspondence.

We then propose to define a transformation function $\beta[\![t]\!]$ from $[\![t]\!](\mathcal{I})$ to $[\![t]\!](\mathcal{J})$, i.e.

$$\beta[\![t]\!] : [\![t]\!](\mathcal{I}) \to [\![t]\!](\mathcal{J}).$$

Again one should regard β as a parameter to $[\![t]\!](\cdots)$ just as $\mathcal{I}$ and $\mathcal{J}$ are. The definition of $\beta[\![t]\!]$ is by induction on the structure of t:

$$\begin{aligned}
&\beta[\![\mathtt{A}_{\rm i}]\!] \equiv \lambda d.d\\
&\beta[\![t_1{\times}t_2]\!] \equiv \lambda(d_1,d_2).\ (\beta[\![t_1]\!](d_1),\beta[\![t_2]\!](d_2))\\
&\beta[\![t_1{\rightarrow}t_2]\!] \equiv \lambda f.\ \lambda p.\ \bigsqcup\{\ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d_1){\sqsubseteq}p\ \}\\
&\beta[\![\underline{\mathtt{A}}_{\rm i}]\!] \equiv \beta_{\rm i}\\
&\beta[\![t_1\underline{\times}t_2]\!] \equiv \beta_{\times}(\beta[\![t_1]\!],\beta[\![t_2]\!])\\
&\beta[\![t_1\underline{\rightarrow}t_2]\!] \equiv \beta_{\rightarrow}(\beta[\![t_1]\!],\beta[\![t_2]\!])
\end{aligned}$$

Several points now need to be addressed. First we must clarify the claims we shall make about the functions $\beta[\![t]\!]$ and the demands that this enforces on the parameter β. Secondly we must find a way of constraining the types t such that the functions $\beta[\![t]\!]$ exist and have the desired properties. Finally, we must show that the definition of $\beta[\![t]\!]$ is as intended, and in particular that it is correct. Closely related to this is the question of why the equation for $\beta[\![t_1{\rightarrow}t_2]\!]$ uses $\sqsubseteq$ and $\bigsqcup$ rather than $\leq[\![\cdots]\!]$ and its associated least upper bound operator $\bigvee$.

3.4.1 Existence

First we need some definitions and simple facts:

Definition 3.4.1. A *representation transformation* (or just a *transformation*) f from a domain $D{=}(D,\sqsubseteq)$ to a domain $E{=}(E,\sqsubseteq)$ is a function that is strict, continuous and compact preserving (see Definition 3.2.2). □

Fact 3.4.2. Let (α,γ) be a pair of adjoined functions between algebraic lattices. Then α is strict and continuous but not necessarily compact preserving. If γ is additionally continuous then α is compact preserving. $\square$

Example 3.4.3. Recall the definition of $\mathbf{S}^{\mathrm{t}}_{\mathrm{int}}$ and $\mathbf{I}^{\mathrm{t}}_{\mathrm{int}}$ in Examples 3.2.10 and 3.2.11. A representation transformation from $\mathbf{S}^{\mathrm{t}}_{\mathrm{int}}$ to $\mathbf{I}^{\mathrm{t}}_{\mathrm{int}}$ may be defined by

$$\lambda d.\begin{cases} + & \text{if } d{>}0 \\ 0 & \text{if } d{=}0 \\ - & \text{if } d{<}0 \\ \perp & \text{if } d{=}\perp \end{cases}$$

(Note that all elements in $\mathbf{S}^{\mathrm{t}}_{\mathrm{int}}$ and $\mathbf{I}^{\mathrm{t}}_{\mathrm{int}}$ are compact.) $\square$

The demands on the paramter β are clarified by:

Definition 3.4.4. A *representation transformer* β from a domain or lattice interpretation $\mathcal{I}$ to a lattice interpretation $\mathcal{J}$ is a specification of:

- representation transformations $\beta_{\mathrm{i}} : \mathcal{I}^{\mathrm{t}}_{\mathrm{i}} \to \mathcal{J}^{\mathrm{t}}_{\mathrm{i}}$,
- operations $\beta_{\times}$ and $\beta_{\to}$ such that

  $$\beta_{\times}(f_1,f_2) : \mathcal{I}^{\mathrm{t}}_{\times}(D_1,D_2) \to \mathcal{J}^{\mathrm{t}}_{\times}(E_1,E_2)$$
  $$\beta_{\to}(f_1,f_2) : \mathcal{I}^{\mathrm{t}}_{\to}(D_1,D_2) \to \mathcal{J}^{\mathrm{t}}_{\to}(E_1,E_2)$$

  are representation transformations whenever $f_{\mathrm{i}} : D_{\mathrm{i}} \to E_{\mathrm{i}}$ are representation transformations, D_1 and D_2 are domains that satisfy the property $\mathcal{I}^{\mathrm{t}}_{\mathrm{P}}$ and E_1 and E_2 are algebraic lattices.

Example 3.4.5. A representation transformer b from the interpretation $\mathbf{S}$ of Example 3.2.10 to the interpretation $\mathbf{I}$ of Example 3.2.11 may be defined by:

- representation transformations $\mathsf{b}_{\mathrm{i}} : \mathbf{S}^{\mathrm{t}}_{\mathrm{i}} \to \mathbf{I}^{\mathrm{t}}_{\mathrm{i}}$

  with $\mathsf{b}_{\mathrm{int}} = \lambda d.\begin{cases} + & \text{if } d{>}0 \\ 0 & \text{if } d{=}0 \\ - & \text{if } d{<}0 \\ \perp & \text{if } d{=}\perp \end{cases}$

  and $\mathsf{b}_{\mathrm{bool}} = \lambda d.\begin{cases} \mathrm{tt} & \text{if } d{=}\mathrm{true} \\ \mathrm{ff} & \text{if } d{=}\mathrm{false} \\ \perp & \text{if } d{=}\perp \end{cases}$

- operations $\mathsf{b}_{\times}$ and $\mathsf{b}_{\to}$ given by

  $$\mathsf{b}_{\times}(f_1,f_2) = \lambda(d_1,d_2).\ (f_1(d_1),f_2(d_2))$$
  $$\mathsf{b}_{\to}(f_1,f_2) = \lambda f.\ \lambda p.\ \bigsqcup\{\ f_2(f(d)) \mid f_1(d)\sqsubseteq p\ \}$$

Here we cannot be more specific about the b_{i} as Example 3.2.11 only is specific about $\mathbf{I}^{\mathrm{t}}_{\mathrm{int}}$ and $\mathbf{I}^{\mathrm{t}}_{\mathrm{bool}}$ but clearly the $\mathsf{b}_{\mathrm{int}}$ and $\mathsf{b}_{\mathrm{bool}}$ exhibited are representation transformations. Assuming that f_1 and f_2 are representation transformations the well-definedness of $\mathsf{b}_{\times}(f_1,f_2)$ is immediate. It is clearly strict and continuous and it preserves compact elements because the compact elements in a cartesian product are the pairs of compact elements in each component. Also $\mathsf{b}_{\rightarrow}(f_1,f_2)$ is well-defined because the least upper bound is taken in an algebraic lattice (called E_2 in Definition 3.4.4). That $\mathsf{b}_{\rightarrow}(f_1,f_2)(f)$ is a continuous function and that $\mathsf{b}_{\rightarrow}(f_1,f_2)$ is a representation transformation is slightly more involved. We omit the details as they follow rather easily from the case $t{=}t_1{\rightarrow}t_2$ in the proof of the following proposition. □

Well-definedness of $\beta[\![t]\!]$ follows from the following fact and proposition:

Fact 3.4.6. If t is a pure type and β a representation transformer then $\beta[\![t]\!]$ is the representation transformation $\lambda d.d$ and $[\![t]\!](\mathcal{I}){=}[\![t]\!](\mathcal{J})$. □

Proposition 3.4.7. The equations for $\beta[\![t]\!]$ define a representation transformation when t is a level-preserving type and β is a representation transformer. □

Proof: We show by structural induction on t that if $\mathbf{lpt}(t)$ then the above equations define a function

$$\beta[\![t]\!] : [\![t]\!](\mathcal{I}) \rightarrow [\![t]\!](\mathcal{J})$$

and that this function is a representation transformation.

The case $t{=}\mathtt{A}_{\mathrm{i}}$ is straightforward. The case $t{=}t_1{\times}t_2$ follows from the induction hypothesis given that the compact elements in a cartesian product $D' \times D''$ are the pairs of compact elements of D' and D'' respectively, i.e. $\mathrm{B}_{D'\times D''} = \mathrm{B}_{D'}{\times}\mathrm{B}_{D''}$. The case $t{=}\underline{\mathtt{A}}_{\mathrm{i}}$ follows from the assumptions on β. In a similar way the cases $t{=}t_1\underline{\times}t_2$ and $t{=}t_1\underline{\rightarrow}t_2$ follow from the assumptions on β and the induction hypothesis.

It remains to consider the case where $t{=}t_1{\rightarrow}t_2$. There are two 'alternatives' in the definition of $\mathbf{lpt}(t_1{\rightarrow}t_2)$ so we first consider the possibility where t_1 is pure and t_2 is level-preserving. We shall write

$$\mathrm{Y}(f,p) = \{\ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d){\sqsubseteq}p\ \}$$

and by Fact 3.4.6 we get $\mathrm{Y}(f,p) = \{\ \beta[\![t_2]\!](f(d)) \mid d{\sqsubseteq}p\ \}$. By continuity of f and $\beta[\![t_2]\!]$ this set contains an upper bound for itself, namely $\beta[\![t_2]\!](f(p))$. Hence $\bigsqcup\mathrm{Y}(f,p)$ always exists and equals $\beta[\![t_2]\!](f(d))$. Next we consider the situation where t_1 is level-preserving and t_2 is impure. Then t_2 is also a lattice type, i.e. $\mathbf{lt}(t_2)$, so that $[\![t_2]\!](\mathcal{J})$ is an algebraic lattice. It is then straightforward that $\bigsqcup\mathrm{Y}(f,p)$ exists.

We have now shown that $\beta[\![t_1 \rightarrow t_2]\!](f)(p)$ always exists when $t_1 \rightarrow t_2$ is level-preserving. To show that $\beta[\![t_1 \rightarrow t_2]\!](f)$ exists we must show that $\bigsqcup \mathrm{Y}(f,p)$ depends continuously on p, i.e. that $\lambda p.\ \bigsqcup \mathrm{Y}(f,p)$ is continuous. First we write

$$\mathrm{Z}(f,p) = \{\ \beta[\![t_2]\!](f(b)) \mid \beta[\![t_1]\!](b) \sqsubseteq p \wedge b \text{ is compact }\}$$

This set has $\bigsqcup \mathrm{Y}(f,p)$ as an upper bound so by consistent completeness $\bigsqcup \mathrm{Z}(f,p)$ exists and we clearly have $\bigsqcup \mathrm{Z}(f,p) \sqsubseteq \bigsqcup \mathrm{Y}(f,p)$. Actually, $\bigsqcup \mathrm{Z}(f,p)$ $= \bigsqcup \mathrm{Y}(f,p)$ as any element d such that $\beta[\![t_1]\!](d) \sqsubseteq p$ may be written as $d = \bigsqcup_{\mathrm{n}} b_{\mathrm{n}}$ where each b_{n} is compact. Then $\mathrm{Z}(f,p)$ contains all $\beta[\![t_2]\!](f(b_{\mathrm{n}}))$ so $\bigsqcup \mathrm{Z}(f,p) \sqsupseteq \bigsqcup_{\mathrm{n}} \beta[\![t_2]\!](f(b_{\mathrm{n}})) = \beta[\![t_2]\!](f(d))$ and hence $\bigsqcup \mathrm{Z}(f,p) \sqsupseteq \bigsqcup \mathrm{Y}(f,p)$ as d was arbitrary.

To show that $\lambda p.\ \bigsqcup \mathrm{Z}(f,p)$ is continuous let $p = \bigsqcup_{\mathrm{n}} p_{\mathrm{n}}$. Clearly $\bigsqcup_{\mathrm{n}} \bigsqcup \mathrm{Z}(f,p_{\mathrm{n}}) \sqsubseteq \bigsqcup \mathrm{Z}(f,p)$ so it suffices to show that $\bigsqcup \mathrm{Z}(f,p) \sqsubseteq \bigsqcup_{\mathrm{n}} \bigsqcup \mathrm{Z}(f,p_{\mathrm{n}})$. If b is compact and $\beta[\![t_1]\!](b) \sqsubseteq \bigsqcup_{\mathrm{n}} p_{\mathrm{n}}$ then by the induction hypothesis $\beta[\![t_1]\!](b)$ is compact so that $\beta[\![t_1]\!](b) \sqsubseteq p_{\mathrm{n}}$ for some n. Hence $\beta[\![t_2]\!](f(b))$ $\sqsubseteq \bigsqcup \mathrm{Z}(f,p_{\mathrm{n}})$ and this shows the result.

Finally, we must show that $\beta[\![t_1 \rightarrow t_2]\!]$ is a representation transformation. So observe that $\beta[\![t_1 \rightarrow t_2]\!](\perp) = \perp$ follows because $\beta[\![t_2]\!]$ is strict. That $\beta[\![t_1 \rightarrow t_2]\!]$ is continuous follows because $\beta[\![t_2]\!]$ is. It now remains to show that $\beta[\![t_1 \rightarrow t_2]\!]$ preserves compact elements. For a domain $D \rightarrow E$, where also D and E are domains, we shall write

$$[d,e] = \lambda d'.\ \begin{cases} e & \text{if } d' \sqsupseteq d \\ \perp & \text{otherwise} \end{cases}$$

The function is continuous if d is compact and is a compact element of $D \rightarrow E$ if additionally e is compact. The general form of a compact element in $D \rightarrow E$ is

$$[b_1,e_1] \sqcup \cdots \sqcup [b_{\mathrm{n}},e_{\mathrm{n}}]$$

(for n$\geq$1) where all b_{i} and e_{i} are compact and we assume that $\{e_{\mathrm{j}} | j \in \mathrm{J}_{d'}\}$ has an upper bound (and by consistent completeness a least upper bound) whenever $d' \in D$ and $\mathrm{J}_{d'} = \{j | b_{\mathrm{j}} \sqsubseteq d'\}$. We shall say that $[b_1,e_1] \sqcup \cdots \sqcup [b_{\mathrm{n}},e_{\mathrm{n}}]$ is a *complete listing* if for all $d' \in D$ there exists $j' \in \{1,\cdots,n\}$ such that $b_{\mathrm{j}'}$ $= \bigsqcup \{b_{\mathrm{j}} | j \in \mathrm{J}_{d'}\}$ and $e_{\mathrm{j}'} = \bigsqcup \{e_{\mathrm{j}} | j \in \mathrm{J}_{d'}\}$. Clearly any compact element can be represented by a complete listing (as the least upper bound of a finite set of compact elements is compact).

For a complete listing $[b_1,e_1] \sqcup \cdots \sqcup [b_{\mathrm{n}},e_{\mathrm{n}}]$ of a compact element in $[\![t_1]\!](\mathcal{I}) \rightarrow [\![t_2]\!](\mathcal{I})$ we now calculate

$$\beta[\![t_1 \rightarrow t_2]\!]([b_1,e_1] \sqcup \cdots \sqcup [b_{\mathrm{n}},e_{\mathrm{n}}]) =$$
$$\lambda p.\ \bigsqcup \{\ \beta[\![t_2]\!](([b_1,e_1] \sqcup \cdots \sqcup [b_{\mathrm{n}},e_{\mathrm{n}}])(b)) \mid \beta[\![t_1]\!](b) \sqsubseteq p \wedge b \text{ is compact }\} =$$

(as the listing is complete and $\beta[\![t_1]\!]$ is strict and continuous)

$$\lambda p.\ \bigsqcup_{\mathrm{j}} \bigsqcup\{\ \beta[\![t_2]\!]([b_{\mathrm{j}},e_{\mathrm{j}}](b))\ |\ \beta[\![t_1]\!](b)\sqsubseteq p\ \wedge\ b \text{ is compact }\} =$$
$$\bigsqcup_{\mathrm{j}} \lambda p.\ \bigsqcup\{\ \beta[\![t_2]\!](e_{\mathrm{j}})\ |\ \beta[\![t_1]\!](b_{\mathrm{j}})\sqsubseteq p\ \} =$$
$$\bigsqcup_{\mathrm{j}} [\beta[\![t_1]\!](b_{\mathrm{j}}),\beta[\![t_2]\!](e_{\mathrm{j}})]$$

By the induction hypothesis all $\beta[\![t_1]\!](b_{\mathrm{j}})$ and $\beta[\![t_2]\!](e_{\mathrm{j}})$ are compact. To show that the above element is compact we must show that $\{\beta[\![t_2]\!](e_{\mathrm{j}})|\mathrm{j}\in\mathrm{J}_{d'}\}$ has an upper bound when $d'\in[\![t_1]\!](\mathcal{J})$ and $\mathrm{J}_{d'} = \{j|\beta[\![t_1]\!](b_{\mathrm{j}})\sqsubseteq d'\}$. From the definition of $\mathbf{lpt}(t_1\rightarrow t_2)$ we know that $\mathbf{it}(t_2)$ or $\mathbf{pt}(t_1)$. If $\mathbf{it}(t_2)$ then $[\![t_2]\!](\mathcal{J})$ is an algebraic lattice so that the set clearly has an upper bound. If $\mathbf{pt}(t_1)$ then $\mathrm{J}_{d'}=\{j|b_{\mathrm{j}}\sqsubseteq d'\}$ so there is $\mathrm{j}'\in\{1,\cdots,n\}$ such that $e_{\mathrm{j}'}= \bigsqcup\{e_{\mathrm{j}}|\mathrm{j}\in\mathrm{J}_{d'}\}$ given the assumption about 'complete listing'. It follows that $\beta[\![t_2]\!](e_{\mathrm{j}'})$ is an upper bound of the set $\{\beta[\![t_2]\!](e_{\mathrm{j}})|j\in\mathrm{J}_{d'}\}$. — This ends the proof of Proposition 3.4.7. □

Remark 3.4.8. The function $\beta[\![t_1\rightarrow t_2]\!]$ is intended as a transformation from the domain $[\![t_1\rightarrow t_2]\!](\mathcal{I})$ to the domain $[\![t_1\rightarrow t_2]\!](\mathcal{J})$. As the reader acquainted with category theory [MacLane, 1971] will know any partially ordered set may be regarded as a simple kind of category. In a similar way a continuous (or at least monotonic) transformation between partially ordered sets may be regarded as a covariant functor. With this in mind we may calculate

$$\begin{aligned}\beta[\![t_1\rightarrow t_2]\!](f) &= \lambda p.\ \bigsqcup\{\ (\beta[\![t_2]\!]\circ f)(d)\ |\ \beta[\![t_1]\!](d)\sqsubseteq p\ \}\\ &= \mathrm{Lan}_{\beta[\![t_1]\!]}(\beta[\![t_2]\!]\circ f)\end{aligned}$$

where we use the formula in [MacLane, 1971, Theorem 4.1, page 236] for the *left Kan extension* of $\beta[\![t_2]\!]\circ f$ along $\beta[\![t_1]\!]$. □

Remark 3.4.9. The first component α of a pair (α,γ) of adjoined functions is often called a *lower adjoint* and the second component γ an *upper adjoint*. In Fact 3.4.2 we said that any lower adjoint is a representation transformation provided we restrict our attention to adjoined pairs of continuouos functions. Assume now that the representation transformer α specifies lower adjoints α_{i} and that $\alpha_{\times}$ and $\alpha_{\rightarrow}$ preserve lower adjoints. Then also $\alpha[\![t]\!]$ will be a lower adjoint whenever t is level-preserving. Writing $\gamma[\![t]\!]$ for the corresponding upper adjoint we then have

$$\alpha[\![t_1\rightarrow t_2]\!](f) = \alpha[\![t_2]\!]\ \circ\ f\ \circ\ \gamma[\![t_1]\!]$$

(Here we have used the fact that an upper adjoint is uniquely determined by its lower adjoint, i.e. if (α,γ_1) and (α,γ_2) are adjoined pairs then $\gamma_1=\gamma_2$.) □

3.4.2 Weak invertibility

To express that $\beta[\![t]\!]$ lives up to the intentions we first need to construct a weak notion of inverse.

Definition 3.4.10. A function $f' : D\times E\rightarrow D$ is a *weak inverse* of a function $f : D\rightarrow E$ and a relation $R : E\times E\rightarrow\{\text{true,false}\}$ if

$$f(d) \sqsubseteq e$$

implies

$$d \sqsubseteq f'(d,e)$$
$$f(f'(d,e))\ R\ e$$

for all $d\in D$ and $e\in E$. □

Here the intention is that f' is the 'inverse' of f, or to be more precise, that (f,f') behaves as much like an adjoined pair of functions as possible. However, D is not (necessarily) an algebraic lattice and so we cannot find a 'best' description of some $e\in E$. Rather we must be content with finding a description $f'(d,e)$ that is 'close' to some $d\in D$ of interest.

We now propose the following definition of a weak inverse $\beta'[\![t]\!]$ of $\beta[\![t]\!]$ and $\leq[\![t]\!]$:

$$\beta'[\![\mathtt{A}_{\mathrm{i}}]\!] \equiv \lambda(d,e).\ e$$
$$\beta'[\![t_1\times t_2]\!] \equiv \lambda((d_1,d_2),(e_1,e_2)).\ (\beta'[\![t_1]\!](d_1,e_1),\ \beta'[\![t_2]\!](d_2,e_2))$$
$$\beta'[\![t_1\rightarrow t_2]\!] \equiv \lambda(f,g).\ \lambda d.\ \beta'[\![t_2]\!](f(d),g(\beta[\![t_1]\!](d)))$$
$$\beta'[\![\underline{\mathtt{A}}_{\mathrm{i}}]\!] \equiv \lambda(d,e).\ d$$
$$\beta'[\![t_1\underline{\times}t_2]\!] \equiv \lambda(d,e).\ d$$
$$\beta'[\![t_1\underline{\rightarrow}t_2]\!] \equiv \lambda(d,e).\ d$$

The behaviour of $\beta'[\![t]\!]$ is easy to characterise in a few special cases:

Fact 3.4.11. If t is pure then $\beta'[\![t]\!](d,e) = e$. □

Fact 3.4.12. If t is impure then $\beta'[\![t]\!](d,e) = d$. □

In the general case we have:

Lemma 3.4.13. The equations for $\beta'[\![t]\!]$ define a weak inverse of $\beta[\![t]\!]$ and $\leq[\![t]\!]$ whenever t is level-preserving. □

Proof: We prove the result by structural induction on t. The case t=$\mathtt{A}_{\mathrm{i}}$ is straightforward as $\beta[\![\mathtt{A}_{\mathrm{i}}]\!] = \lambda d.d$ and $\leq[\![\mathtt{A}_{\mathrm{i}}]\!]\equiv=$. The case t=$t_1\times t_2$ follows from the induction hypothesis. The case t=$\underline{\mathtt{A}}_{\mathrm{i}}$ is straightforward as $\leq[\![\underline{\mathtt{A}}_{\mathrm{i}}]\!]\equiv\sqsubseteq$. Also the cases t=$t_1\underline{\times}t_2$ and t=$t_1\underline{\rightarrow}t_2$ are straightforward as we have $\leq[\![t]\!]\equiv\sqsubseteq$.

It remains to consider the case t=$t_1\rightarrow t_2$. So assume that $\beta[\![t_1\rightarrow t_2]\!](f)\sqsubseteq g$, i.e.

$$\beta[\![t_1]\!](d) \sqsubseteq e \Rightarrow \beta[\![t_2]\!](f(d)) \sqsubseteq g(e) \qquad (*)$$

for all d and e. To show $f \sqsubseteq \beta'[\![t_1 \rightarrow t_2]\!](f,g)$ we consider an argument d and must show

$$f(d) \sqsubseteq \beta'[\![t_2]\!](f(d),g(\beta[\![t_1]\!](d)))$$

and this follows from the induction hypothesis given the assumption (*). To show that

$$\beta[\![t_1 \rightarrow t_2]\!](\beta'[\![t_1 \rightarrow t_2]\!](f,g)) \leq[\![t_1 \rightarrow t_2]\!]\ g$$

we let

$$e \leq[\![t_1]\!]\ e'$$

and must show

$$\bigsqcup\{\ \beta[\![t_2]\!](\beta'[\![t_1 \rightarrow t_2]\!](f,g)(d)) \mid \beta[\![t_1]\!](d) \sqsubseteq e\ \} \leq[\![t_2]\!]\ g(e')$$

i.e.

$$\bigsqcup\{\ \beta[\![t_2]\!](\beta'[\![t_2]\!](f(d),g(\beta[\![t_1]\!](d)))) \mid \beta[\![t_1]\!](d) \sqsubseteq e\ \} \leq[\![t_2]\!]\ g(e')$$

We now consider the 'alternatives' in the definition of $\mathbf{lpt}(t_1 \rightarrow t_2)$ one by one. If t_1 is pure the inequality reduces to

$$\beta[\![t_2]\!](\beta'[\![t_2]\!](f(e),g(e))) \leq[\![t_2]\!]\ g(e)$$

as e=e'. The desired result then follows from the induction hypothesis. If t_2 is impure the inequality reduces to

$$\beta[\![t_1]\!](d) \sqsubseteq e \Rightarrow \beta[\![t_2]\!](\beta'[\![t_2]\!](f(d),g(\beta[\![t_1]\!](d)))) \sqsubseteq g(e')$$

So assume that $\beta[\![t_1]\!](d) \sqsubseteq e$. Using (*) and the induction hypothesis for t_2 we then get

$$\beta[\![t_2]\!](\beta'[\![t_2]\!](f(d),g(\beta[\![t_1]\!](d)))) \sqsubseteq g(\beta[\![t_1]\!](d))$$

The result then follows as $\beta[\![t_1]\!](d) \sqsubseteq e \sqsubseteq e'$. — This ends the proof of Lemma 3.4.13. □

The main point of the above lemma is to establish the following corollary showing that another definition of $\beta[\![t_1 \rightarrow t_2]\!]$ is possible. However, as is evidenced by [Nielson, 1988a] the present route presents fewer technical complications.

Corollary 3.4.14. $\beta[\![t_1 \rightarrow t_2]\!](f) = \lambda p.\ \bigvee \{\beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d) \leq[\![t_1]\!]\ p\}$ whenever $t_1 \rightarrow t_2$ is level-preserving and $\bigvee$ denotes the least upper bound operator wrt. $\leq[\![t_2]\!]$. □

Proof: We shall write

$$\mathrm{X}(f,p) = \{\ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d) \leq[\![t_1]\!]\ p\ \}$$

and recall the definition of $\mathrm{Y}(f,p)$ given in the proof of Proposition 3.4.7. As $\mathrm{X}(f,p) \subseteq \mathrm{Y}(f,p)$ and $\bigsqcup\mathrm{Y}(f,p)$ exists we get that $\bigsqcup\mathrm{X}(f,p)$ exists and $\bigsqcup\mathrm{X}(f,p) \sqsubseteq \bigsqcup\mathrm{Y}(f,p)$. Next let $\beta[\![t_1]\!](d)\sqsubseteq p$ and note that by setting $d' = \beta'[\![t_1]\!](d,p)$ we have $d \sqsubseteq d'$ and $\beta[\![t_1]\!](d') \leq[\![t_1]\!]\ p$. Hence

$$\beta[\![t_2]\!](f(d)) \sqsubseteq \beta[\![t_2]\!](f(d')) \sqsubseteq \bigsqcup\mathrm{X}(f,p)$$

so that $\bigsqcup\mathrm{Y}(f,p) \sqsubseteq \bigsqcup\mathrm{X}(f,p)$ and thus $\bigsqcup\mathrm{Y}(f,p) = \bigsqcup\mathrm{X}(f,p)$.

Next we shall show that $\bigsqcup\mathrm{X}(f,p)$ is the least upper bound of $\mathrm{X}(f,p)$ wrt. $\leq[\![t_2]\!]$. For this we consider the 'alternatives' in the definition of $\mathbf{lpt}(t_1 \rightarrow t_2)$ one by one. If t_1 is pure the set $\mathrm{X}(f,p)$ equals the singleton $\{\beta[\![t_2]\!](f(p))\}$ and clearly $\beta[\![t_2]\!](f(p))$ is the least upper bound of this set wrt. the suborder $\leq[\![t_2]\!]$. If t_2 is impure the suborder $\leq[\![t_2]\!]$ amounts to $\sqsubseteq$ and then clearly $\bigsqcup\mathrm{X}(f,p)$ is the least upper bound of $\mathrm{X}(f,p)$ wrt. $\sqsubseteq$. $\square$

3.4.3 Optimality

It remains to demonstrate that the transformation $\beta[\![t]\!]$ has the required properties (whenever t is a level-preserving type). We express the correctness using a relation $\mathcal{R}[\![t]\!]$ as defined in the previous subsection and clearly β and $\mathcal{R}$ have to 'cooperate':

Definition 3.4.15. An admissible relation $R : D \times E \rightarrow \{\text{true,false}\}$ *cooperates with* a representation transformation $f : D \rightarrow E$ and an admissible relation $R' : E \times E \rightarrow \{\text{true,false}\}$ if

$$R(d,p) \equiv R'(f(d),p)$$

A correctness correspondence $\mathcal{R}$ *cooperates with* a representation transformer β if

- $\mathcal{R}_{\mathrm{i}}$ cooperates with β_{i} and $\sqsubseteq$, and
- if R_{i} cooperates with f_{i} and $\sqsubseteq$ (for i=1,2) then

  $\mathcal{R}_{\times}(R_1,R_2)$ cooperates with $\beta_{\times}(f_1,f_2)$ and $\sqsubseteq$, and
  $\mathcal{R}_{\rightarrow}(R_1,R_2)$ cooperates with $\beta_{\rightarrow}(f_1,f_2)$ and $\sqsubseteq$

Example 3.4.16. The correctness correspondence cor of Example 3.3.4 cooperates with the representation transformer b of Example 3.4.5. $\square$

Lemma 3.4.17. If $\mathcal{R}$ cooperates with β then

$$\mathcal{R}[\![t]\!] \text{ cooperates with } \beta[\![t]\!] \text{ and } \leq[\![t]\!]$$

for all level-preserving types t. $\square$

Proof: We must prove $\mathcal{R}[\![t]\!](d,p) \equiv (\beta[\![t]\!](d) \leq[\![t]\!]\ p)$ by structural induction on a level-preserving type t.

The case t=$\mathtt{A_i}$ is straightforward as $\mathcal{R}[\![t]\!]$≡=, $\beta[\![t]\!]=\lambda d.d$ and $\leq[\![t]\!]$≡=. The case t=$t_1 \times t_2$ follows from the induction hypothesis and the componentwise definitions of $\mathcal{R}[\![t_1 \times t_2]\!]$, $\beta[\![t_1 \times t_2]\!]$ and $\leq[\![t_1 \times t_2]\!]$. The case t=$\underline{\mathtt{A}}_{\mathtt{i}}$ is immediate from the assumptions. The cases t=$t_1 \underline{\times} t_2$ and t=$t_1 \underline{\rightarrow} t_2$ are straightforward given the assumptions, the induction hypothesis and Lemma 3.3.7.

It remains to consider the case t=$t_1 \rightarrow t_2$. We first assume that $\mathcal{R}[\![t_1 \rightarrow t_2]\!](f,g)$ and by the induction hypothesis this amounts to

$$\beta[\![t_1]\!](d) \leq[\![t_1]\!]\ e \Rightarrow \beta[\![t_2]\!](f(d)) \leq[\![t_2]\!]\ g(e) \qquad (*)$$

for all d and e. To show $\beta[\![t_1 \rightarrow t_2]\!](f) \leq[\![t_1 \rightarrow t_2]\!]\ g$ we assume that $e \leq[\![t_1]\!]$ e' and must show

$$\bigvee \{\ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d) \leq[\![t_1]\!]\ e\ \} \leq[\![t_2]\!]\ g(e')$$

where we have used Corollary 3.4.14. For this it suffices to show that

$$\beta[\![t_1]\!](d) \leq[\![t_1]\!]\ e \Rightarrow \beta[\![t_2]\!](f(d)) \leq[\![t_2]\!]\ g(e')$$

and this follows from (*) as $g \leq[\![t_1 \rightarrow t_2]\!]\ g$ implies $g(e) \leq[\![t_2]\!]\ g(e')$. Next we assume that $\beta[\![t_1 \rightarrow t_2]\!](f) \leq[\![t_1 \rightarrow t_2]\!]\ g$, i.e. that

$$\mathrm{e} \leq[\![t_1]\!]\ e' \Rightarrow \bigvee \{\ \beta[\![t_2]\!](f(d)) \mid \beta[\![t_1]\!](d) \leq[\![t_1]\!]\ e\ \} \leq[\![t_2]\!]\ g(e')$$

holds for all e and e'. It follows that

$$\beta[\![t_1]\!](d) \leq[\![t_1]\!]\ e \wedge e \leq[\![t_1]\!]\ e' \Rightarrow \beta[\![t_2]\!](f(d)) \leq[\![t_2]\!]\ g(e')$$

and by choosing e'=e and using the induction hypothesis we have

$$\mathcal{R}[\![t_1]\!](d,e) \Rightarrow \mathcal{R}[\![t_2]\!](f(d),g(e))$$

for all d and e. But this amounts to $\mathcal{R}[\![t_1 \rightarrow t_2]\!](f,g)$. $\square$

3.4.4 Summary

Let $\beta : \mathcal{I} \rightarrow \mathcal{J}$, or $\beta : \mathcal{I}^{\mathrm{t}} \rightarrow \mathcal{J}^{\mathrm{t}}$ to be precise, be a representation transformer and let $\hat{\beta}$ be a correctness correspondence that cooperates with β. An example was given in Example 3.4.16 and we should stress that we do not claim that $\hat{\cdot}$ is a function as it does not seem possible to define $\hat{\beta}_{\times}$ and $\hat{\beta}_{\rightarrow}$ from $\beta_{\times}$ and $\beta_{\rightarrow}$ in general.

If $\mathcal{I}$ is an interpretation for TML[**lpt**,**dt**] we may define an interpretation $\beta(\mathcal{I})$ for TML[**lpt**,**dt**], called the *induced analysis*, as follows:

- $(\beta(\mathcal{I}))^{\mathrm{t}} = \mathcal{J}^{\mathrm{t}}$,
- for each basic expression or combinator ψ used with type t_ψ: $(\beta(\mathcal{I}))^{\mathrm{e}}_{\psi} = \beta[\![t_\psi]\!](\mathcal{I}^{\mathrm{e}}_{\psi})$

We can now be assured that

- $\beta(\mathcal{I})$ is correct, i.e. $\mathcal{I}\ \hat{\beta}\ \beta(\mathcal{I})$,
- $\beta(\mathcal{I})$ is optimal, i.e. $\mathcal{I}\ \hat{\beta}\ \mathcal{J} \Rightarrow \beta(\mathcal{I}) \leq \mathcal{J}$

In both cases we use that $\mathcal{I}\ \hat{\beta}\ \mathcal{K}$ amounts to $\beta(\mathcal{I}) \leq \mathcal{K}$ given that $\hat{\beta}$ cooperates with β and that $\leq$ is a partial order when we restrict our attention to interpretations of TML[**lpt**,**dt**].

3.5 Expected Forms of Analyses

Even though the induced analyses of the previous subsection are optimal they are not necessarily in a form where they are practical or even computable. As an example consider composition $\Box$ which is interpreted as $\mathbf{S}^{\mathrm{e}}_{\Box} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$ in the lazy standard semantics. The induced version is

$$\begin{array}{l}
\beta[\![(t\text{'}\underline{\rightarrow}t\text{''})\rightarrow(t\underline{\rightarrow}t\text{'})\rightarrow(t\underline{\rightarrow}t\text{''})]\!]\ (\mathbf{S}^{\mathrm{e}}_{\Box}) = \\
\lambda vi_1.\ \bigsqcup\{\beta[\![(t\underline{\rightarrow}t\text{'})\rightarrow(t\underline{\rightarrow}t\text{''})]\!]\ (\mathbf{S}^{\mathrm{e}}_{\Box}(vs_1)) \mid \beta[\![t\text{'}\underline{\rightarrow}t\text{''}]\!](vs_1)\sqsubseteq vi_1\} = \\
\lambda vi_1.\ \ \lambda vi_2.\ \ \bigsqcup\{\beta[\![t\underline{\rightarrow}t\text{''}]\!]\ \ (\mathbf{S}^{\mathrm{e}}_{\Box}(vs_1)(vs_2)) \mid \beta[\![t\text{'}\underline{\rightarrow}t\text{''}]\!](vs_1)\sqsubseteq vi_1 \ \wedge \\
\beta[\![t\underline{\rightarrow}t\text{'}]\!](vs_2)\sqsubseteq vi_2\} = \\
\lambda vi_1.\ \lambda vi_2.\ \bigsqcup\{\beta[\![t\underline{\rightarrow}t\text{''}]\!]\ (vs_1\circ vs_2) \mid \beta[\![t\text{'}\underline{\rightarrow}t\text{''}]\!](vs_1)\sqsubseteq vi_1 \wedge \beta[\![t\underline{\rightarrow}t\text{'}]\!](vs_2)\sqsubseteq vi_2\}
\end{array}$$

and this is not as easy to implement as one could have hoped for. In the special case where β specifies lower adjoints as in Remark 3.4.9 we have a slightly nicer induced version (writing α for β):

$$\begin{array}{l}
\alpha[\![(t\text{'}\underline{\rightarrow}t\text{''})\rightarrow(t\underline{\rightarrow}t\text{'})\rightarrow(t\underline{\rightarrow}t\text{''})]\!]\ (\mathbf{S}^{\mathrm{e}}_{\Box}) = \\
\lambda vi_1.\ \alpha[\![(t\underline{\rightarrow}t\text{'})\rightarrow(t\underline{\rightarrow}t\text{''})]\!]\ (\mathbf{S}^{\mathrm{e}}_{\Box}(\gamma[\![t\text{'}\underline{\rightarrow}t\text{''}]\!](vi_1))) = \\
\lambda vi_1.\ \lambda vi_2.\ \alpha[\![t\underline{\rightarrow}t\text{''}]\!]\ (\mathbf{S}^{\mathrm{e}}_{\Box}(\gamma[\![t\text{'}\underline{\rightarrow}t\text{''}]\!](vi_1))(\gamma[\![t\underline{\rightarrow}t\text{'}]\!](vi_2)))
\end{array}$$

If we assume that $\alpha_{\rightarrow}$ is as in Example 3.4.5, i.e.

$$\alpha_{\rightarrow}(f_1,f_2) = \lambda f.\ \lambda p.\ \bigsqcup\{\ f_2(f(d)) \mid f_1(d)\sqsubseteq p\ \}$$

then it follows much as in Remark 3.4.9 that $\alpha[\![t\underline{\rightarrow}t\text{''}]\!](f) = \alpha[\![t\text{''}]\!]\circ f\circ\gamma[\![t]\!]$, $\gamma[\![t\text{'}\underline{\rightarrow}t\text{''}]\!](f) = \gamma[\![t\text{''}]\!]\circ f\circ\alpha[\![t\text{'}]\!]$ and $\gamma[\![t\underline{\rightarrow}t\text{'}]\!](f) = \gamma[\![t\text{'}]\!]\circ f\circ\alpha[\![t]\!]$. It follows that

$$
\begin{aligned}
&\alpha\llbracket(t'\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')\rrbracket\,(\mathbf{S}\overset{e}{\Box}) = \\
&\lambda vi_1.\ \lambda vi_2.\ \alpha\llbracket t''\rrbracket \circ (\gamma\llbracket t''\rrbracket\circ vi_1\circ\alpha\llbracket t'\rrbracket) \circ (\gamma\llbracket t'\rrbracket\circ vi_2\circ\alpha\llbracket t\rrbracket) \circ \gamma\llbracket t\rrbracket = \\
&\lambda vi_1.\ \lambda vi_2.\ (\alpha\llbracket t''\rrbracket\circ\gamma\llbracket t''\rrbracket) \circ vi_1 \circ (\alpha\llbracket t'\rrbracket\circ\gamma\llbracket t'\rrbracket) \circ vi_2 \circ (\alpha\llbracket t\rrbracket\circ\gamma\llbracket t\rrbracket)
\end{aligned}
$$

but as $\alpha\llbracket\cdots\rrbracket\circ\gamma\llbracket\cdots\rrbracket$ in general will differ from the identity also this is not as easy to implement as one could have hoped for.

What we shall do instead is to use functional composition in all analyses. This may not be as precise as possible but will be easier to implement and, as we shall show below, will indeed be correct and thus will make applications easier. A similar treatment can be given for the other combinators and we shall consider a few examples in this subsection. Additional examples may be found in [Nielson, 1989,Nielson, 1986b].

Example 3.5.1. For a lattice interpretation $\mathcal{J}$ of TML[**lpt**,**dt**] we suggest modelling the composition combinator $\Box$ as functional composition, i.e.

$$\mathcal{J}\overset{e}{\Box} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$$

and we shall say that this is the *expected form* of $\Box$. Actually the expected form depends on whether we consider forward or backward analyses, i.e. whether $\underline{\rightarrow}$ is interpreted as $\rightarrow$ or as $\leftarrow$, as is illustrated by Examples 3.2.11 and 3.2.13 but in this subsection we shall only consider forward analyses.

To demonstrate the correctness of this expected form let $\mathcal{I}$ be a domain or lattice interpretation with $\mathcal{I}\overset{e}{\Box} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(v_2(w))$. Here $\mathcal{I}$ may be thought of as the standard semantics or some other lattice interpretation that uses the expected form for $\Box$. We shall then show

$$
\begin{aligned}
&\beta\llbracket(t'\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')\rrbracket(\mathcal{I}\overset{e}{\Box}) \\
&\leq\llbracket(t'\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')\rrbracket \\
&\mathcal{J}\overset{e}{\Box}
\end{aligned}
$$

where $\beta : \mathcal{I} \rightarrow \mathcal{J}$ is a representation transformer. In analogy with the assumption that $\underline{\rightarrow}$ is interpreted as $\rightarrow$ in both $\mathcal{I}$ and $\mathcal{J}$ it is natural to assume that $\beta_{\rightarrow}$ is as in Example 3.4.5, i.e.

$$\beta_{\rightarrow}(f_1,f_2) = \lambda f.\ \lambda p.\ \bigsqcup\,\{\ f_2(f(d)) \mid f_1(d)\sqsubseteq p\ \}$$

As $(t'\underline{\rightarrow}t'')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t'')$ is level-preserving it follows that t, t' and t'' are impure so that $\leq\llbracket t\rrbracket\equiv\sqsubseteq$, $\leq\llbracket t'\rrbracket\equiv\sqsubseteq$ and $\leq\llbracket t''\rrbracket\equiv\sqsubseteq$. The desired result then amounts to assuming that

$$
\begin{aligned}
&\beta\llbracket t'\rrbracket(wi) \sqsubseteq wj \Rightarrow \beta\llbracket t''\rrbracket(vi_1(wi)) \sqsubseteq vj_1(wj) \\
&\beta\llbracket t\rrbracket(wi) \sqsubseteq wj \Rightarrow \beta\llbracket t'\rrbracket(vi_2(wi)) \sqsubseteq vj_2(wj) \\
&\beta\llbracket t\rrbracket(wi) \sqsubseteq wj
\end{aligned}
$$

and showing that

$$\beta[\![t"]\!](vi_1(vi_2(wi)) \sqsubseteq vj_1(vj_2(wj))$$

and this is straightforward. □

Example 3.5.2. For a lattice interpretation $\mathcal{J}$ of TML[**lpt**,**dt**] we suggest that the *expected form* of `Fix` amounts to a finite, say 27, number of iterations starting with the 'most imprecise' property $\top$, i.e.

$$\mathcal{J}^{\mathrm{e}}_{\mathrm{Fix}} = \lambda v.\lambda w.\ (v(w))^{27}(\top)$$

(Recall that if `Fix` is used with type $(t\underline{\rightarrow}t'\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')$ then t' will be an impure type so that $[\![t']\!](\mathcal{J})$ is an algebraic lattice and thus $\top=\bigsqcup[\![t']\!](\mathcal{J})$ does exist.) This choice of expected form differs from the choice made in the standard semantics of Example 3.2.10 (or the detection of signs analysis of Example 3.2.11) and is motivated by the desire to ensure that an analysis by abstract interpretation terminates. However, it means that we have to give seperate proofs for the correctness of this expected form with respect to the standard semantics and for the correctness of continuing to use this expected form.

So let $\mathcal{I}$ be a domain or lattice interpretation with $\mathcal{I}^{\mathrm{e}}_{\mathrm{Fix}} = \lambda v.\ \lambda w.$ FIX$(v(w))$. Let $\beta : \mathcal{I} \rightarrow \mathcal{J}$ be a representation transformer with $\beta_{\rightarrow}$ as in the previous example. We must then show

$$\begin{array}{l}\beta[\![(t\underline{\rightarrow}t'\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!](\lambda v.\lambda w.\mathrm{FIX}(v(w)))\\ \leq[\![(t\underline{\rightarrow}t'\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!]\\ (\lambda v.\lambda w.(v(w))^{27}(\top))\end{array}$$

where again t and t' will be impure so that $\leq[\![t]\!]$ and $\leq[\![t']\!]$ both equal $\sqsubseteq$. This amounts to assuming

$$\begin{array}{l}\beta[\![t]\!](wi_1) \sqsubseteq wj_1 \wedge \beta[\![t']\!](wi_2) \sqsubseteq wj_2 \Rightarrow\\ \qquad\beta[\![t']\!](vi(wi_1)(wi_2)) \sqsubseteq vj(wj_1)(wj_2)\end{array}$$

$$\beta[\![t]\!](wi) \sqsubseteq wj$$

and showing

$$\beta[\![t']\!](\mathrm{FIX}(vi(wi))) \sqsubseteq (vj(wj))^{\mathrm{m}}(\top)$$

for m=27. We shall prove this by induction on m and the base case m=0 is immediate. The induction step then follows from the induction hypothesis and the fact that $(vi(wi))(\mathrm{FIX}(vi(wi)))$ equals $(\mathrm{FIX}(vi(wi)))$.

Next let $\mathcal{K}$ be a lattice interpretation that uses the expected form for `Fix` and let $\beta : \mathcal{J} \rightarrow \mathcal{K}$ be a representation transformer with $\beta_{\rightarrow}$ as above. We must then show

$$\begin{array}{l}\beta[\![(t\underline{\rightarrow}t'\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!](\lambda v.\lambda w.(v(w))^{27}(\top))\\ \leq[\![(t\underline{\rightarrow}t'\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!]\\ (\lambda v.\lambda w.(v(w))^{27}(\top))\end{array}$$

This amounts to assuming

$$\beta[\![t]\!](wj_1) \sqsubseteq wk_1 \wedge \beta[\![t']\!](wj_2) \sqsubseteq wk_2 \Rightarrow$$
$$\beta[\![t']\!](vj(wj_1)(wj_2)) \sqsubseteq vk(wk_1)(wk_2)$$

$$\beta[\![t]\!](wj) \sqsubseteq wk$$

and showing

$$\beta[\![t']\!]((vj(wj))^{\mathrm{m}}(\top)) \sqsubseteq (vk(wk))^{\mathrm{m}}(\top)$$

for m=27. This is once again by induction on m. □

Example 3.5.3. In TML[**lpt**,**dt**] the meaning of `fix` remains constant, i.e. `fix` is not interpreted by an interpretation but always amounts to the least fixed point FIX. Consider now a version of TML where the meaning of `fix` is not constant and thus must be given by an interpretation. When `fix` is used with type $(t \rightarrow t) \rightarrow t$ this means that we will have to restrict $(t \rightarrow t) \rightarrow t$ to be level-preserving. It is straightforward to verify that this means that t must either be pure or impure. When t is pure it is natural to let an interpreation $\mathcal{J}$ use the expected form

$$\mathcal{J}^{\mathrm{e}}_{\mathrm{fix}} = \lambda v.\ \mathrm{FIX}(v)$$

whereas when t is impure it is natural to let $\mathcal{J}$ use the expected form

$$\mathcal{J}^{\mathrm{e}}_{\mathrm{fix}} = \lambda v.\ v^{27}(\top).$$

The correctness of this is shown in [Nielson, 1989]. □

Example 3.5.4. For a lattice interpretation $\mathcal{J}$ of TML[**lpt**,**dt**] we suggest that the *expected form* of `If` amounts to simply combining the effects of the true and else branches, i.e.

$$\mathcal{J}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ (v_2(w)) \sqcup (v_3(w))$$

This is slightly coarser than what we did in Examples 3.2.11 and 3.2.14 in that v_1 is not taken into account but this is in agreement with common practice in data flow analysis [Aho, Sethi and Ullman, 1986].

To show the correctness of this let $\mathcal{I}$ be a domain interpretation along the lines of the lazy standard semantics, i.e.

$$\mathcal{I}^{\mathrm{t}}_{\mathrm{bool}} = A_{\mathrm{bool}}$$

$$\mathcal{I}^{\mathrm{e}}_{\mathrm{If}} = \lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ \begin{cases} v_2(w) & \text{if } v_1(w)\text{=true} \\ v_3(w) & \text{if } v_1(w)\text{=false} \\ \perp & \text{if } v_1(w)\text{=}\perp \end{cases}$$

(as well as $\mathcal{I}^{\mathrm{t}}_{\underline{\rightarrow}}{=}\rightarrow$ as is the case for all interpretations considered in this subsection). Also let $\beta : \mathcal{I} \rightarrow \mathcal{J}$ be a representation transformer with $\beta_{\rightarrow}$ as in the previous example. We must then show

$$\begin{aligned}&\beta[\![(t\underline{\rightarrow}\texttt{Bool})\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!]\ (\mathcal{I}^{\mathrm{e}}_{\mathrm{If}})\\ &\leq[\![(t\underline{\rightarrow}\texttt{Bool})\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!]\\ &(\mathcal{J}^{\mathrm{e}}_{\mathrm{If}})\end{aligned}$$

and this follows much as in Example 3.3.4 so we dispense with the details.

Next let $\mathcal{K}$ be a lattice interpretation that also uses the expected form for `If`. Then we must show

$$\begin{aligned}&\beta[\![(t\underline{\rightarrow}\texttt{Bool})\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!]\ (\lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ (v_2(w))\\ &\sqcup (v_3(w)))\\ &\leq[\![(t\underline{\rightarrow}\texttt{Bool})\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')\rightarrow(t\underline{\rightarrow}t')]\!]\\ &(\lambda v_1.\ \lambda v_2.\ \lambda v_3.\ \lambda w.\ (v_2(w)) \sqcup (v_3(w)))\end{aligned}$$

and for this we shall need to assume that β specifies lower adjoints as in Remark 3.4.9. Then $\beta[\![t']\!]$ is (binary) additive and it is then straightforward to show the desired result. □

Example 3.5.5. For a lattice interpretation $\mathcal{J}$ of TML[**lpt**,**dt**] the expected forms of the combinators `Tuple`, `Fst` and `Snd` depend on how $\underline{\times}$ is interpreted. When $\underline{\times}$ is the tensor product $\otimes$ one may suggest expected forms based on the definitions given Example 3.2.14 but as we have not covered the tensor product in any detail we shall not look further into this here. We thus concentrate on the case where $\underline{\times}$ is interpreted as cartesian product and here we suggest

$$\begin{aligned}\mathcal{J}^{\mathrm{e}}_{\mathrm{Tuple}} &= \lambda v_1.\ \lambda v_2.\ \lambda w.\ (v_1(w),v_2(w))\\ \mathcal{J}^{\mathrm{e}}_{\mathrm{Fst}} &= \lambda v.\ \lambda w.\ d_1 \text{ where } (d_1,d_2){=}v(w)\\ \mathcal{J}^{\mathrm{e}}_{\mathrm{Snd}} &= \lambda v.\ \lambda w.\ d_2 \text{ where } (d_1,d_2){=}v(w)\end{aligned}$$

For correctness we shall assume that $\mathcal{I}$ is a domain or lattice interpretation that uses 'analogous definitions', either because it is the standard semantics, or because it is some lattice interpretation that also uses the expected forms. Furthermore we shall assume that $\beta : \mathcal{I} \rightarrow \mathcal{J}$ is a representation transformer with $\beta_{\rightarrow}$ as in the previous examples and with $\beta_{\times}$ as in Example 3.4.5, i.e.

$$\beta_{\times}(f_1,f_2) = \lambda(d_1,d_2).\ (f_1(d_1),f_2(d_2))$$

The correctness is then expressed and proved using the pattern of the previous examples and we omit the details. □

Example 3.5.6. For a lattice interpretation $\mathcal{J}$ of TML[**lpt**,**dt**] we shall suggest the following expected forms for `Curry` and `Apply`:

$$\mathcal{J}^{\mathrm{e}}_{\mathrm{Curry}} = \lambda v.\ \lambda w_1.\ \lambda w_2.\ v(w_1,w_2)$$
$$\mathcal{J}^{\mathrm{e}}_{\mathrm{Apply}} = \lambda v_1.\ \lambda v_2.\ \lambda w.\ v_1(w)(v_2(w))$$

Here we assume that $\underline{\rightarrow}$ is interpreted as $\rightarrow$ and that $\underline{\times}$ is interpreted as cartesian product $\times$. To show correctness we consider a domain or lattice interpretation $\mathcal{I}$ that uses 'analogous definitions'. Concerning the representation transformer $\beta : \mathcal{I} \rightarrow \mathcal{J}$ we shall assume that $\beta_{\rightarrow}$ and $\beta_{\times}$ are as in the previous example. The correctness is then expressed and proved using the pattern of the previous examples and we omit the details. □

These examples have illustrated that the practical application of a framework for abstract interpretation may be facilitated by studying certain expected forms for the combinators. In this way one obtains correct and implementable analyses than can be arranged always to terminate although at the expense of obtaining more imprecise results than those guaranteed by the induced analyses.

3.6 Extensions and Limitations

In this section we have aimed at demonstrating the main approach, the main definitions, the main theorems and the main proof techniques employed in a framework for abstract interpretation. To do so we have used a rather small metalanguage based on the typed λ-calculus and in this subsection we conclude by discussing possible extensions and current limitations.

The discussion will center around [Nielson, 1989]. The metalanguage $\mathrm{TML_m}$ considered there has sum types and recursive types in addition to the base types, product types and function types. However, the well-formedness conditions imposed on the two-level types in [Nielson, 1989] are somewhat more demanding than those imposed here. Let $\mathbf{mc}(t)$ denote the condition that every underlined type constructor in t has arguments that are all-underlined and that any underlined construct occurs in a subtype of the form $t'\underline{\rightarrow}t''$. Then the fragment of $\mathrm{TML_m}$ [Nielson, 1989] that only has base types, product types and function types corresponds to TML[$\mathbf{lpt}\wedge\mathbf{mc}$,$\mathbf{dt}\wedge\mathbf{mc}$][7] rather than the TML[$\mathbf{lpt}$,$\mathbf{dt}$] studied here. As an example this means that a type like $(\mathtt{A}_{\mathrm{int}}\rightarrow\underline{\mathtt{A}}_{\mathrm{int}})\underline{\times}\underline{\mathtt{A}}_{\mathrm{bool}}$ is well-formed in TML[$\mathbf{lpt}$,$\mathbf{dt}$] but not in TML[$\mathbf{lpt}\wedge\mathbf{mc}$,$\mathbf{dt}\wedge\mathbf{mc}$].

Concerning sum types one can perform a development close to that performed for product types. In a sense an analogue to the relational method is obtained by modelling $\underline{+}$ as cartesian product whereas an analogue of the independent attribute method is obtained by interpreting $\underline{+}$ as a kind of sum (adapted to produce an algebraic lattice). For recursive types there are various ways of solving the recursive type equations and one of these

[7] One may verify that $\mathbf{dt}\wedge\mathbf{mc}$ is equivalent to $\mathbf{mc}$.

amounts to truncating a recursive structure at a fixed depth. Turning to expressions an analysis like detection of signs is formulated and proved correct for the whole metalanguage and also strictness analysis can be handled [Nielson, 1988a]. It is also shown that induced analyses exist in general and this is used to give a characterisation of the role of the *collecting semantics* (*accumulating standard semantics* in the terminology of the Glossary). For a second-order and backward analysis like live variables analysis a formulation much like the one given in Example 3.2.13 is proved correct. As we already said in Subsection 3.1 this may be extended with several *versions* of underlined type constructors without a profound change in the theory.

Even for TML[**lpt**$\wedge$**mc**,**dt**$\wedge$**mc**] the development in [Nielson, 1989] goes a bit further than overviewed here. The presence of the constraint **mc** makes it feasible to study interpreations, so-called *frontier interpretations*, where the interpretation of underlined types is *not* specified in a structural way. (In a sense one considers a version of the metalanguage without $\underline{\times}$ and $\underline{\rightarrow}$ but with a greatly expanded index set I' over which the index i in $\underline{\mathtt{A}}_{\mathtt{i}}$ ranges.) This makes is feasible to give a componentwise definition of the *composition* β'$\circ\beta$ of frontier representation transformers. This is of particular interest when β' only specifies representation transformations that are lower adjoints. Writing β'$=\alpha$ one can then show that $(\alpha\circ\beta)[\![t]\!]$ $= \alpha[\![t]\!]\circ\beta[\![t]\!]$ for level-preserving types t and this may be regarded as the basis for developing abstract interpretations in a stepwise manner. Also the transformation from β to $\hat{\beta}$ becomes functional and one can show that $\widehat{\alpha \circ \beta} = \hat{\beta} \circ \hat{\alpha}$.

From a practical point of view a main limitation is that we have not incorporated *stickiness*, i.e. we transform a property through a program but we do not record the properties that reach a given program point. Such a development would be very desirable when one wants to exploit the results of an analysis to enable *program transformations* that are not valid, i.e. meaning preserving, in general. This is illustrated in [Nielson, 1985a] that uses the results of analyses as specified in [Nielson, 1982]. It applies equally well to the flow analysis techniques used in practical compilers.

However, it is not a minor task to incorporate this in the present framework. One reason is that the way it is done depends heavily on details of the *evaluation order* used in an implementation and these details are not specified by the standard semantics.

4 Other Analyses, Language Properties, and Language Types

In the first sections we described the roots of abstract interpretation and gave a motivated development of the Cousot approach. We then showed how their methods can be systematically extended to languages defined by denotational semantics, using logical relations to lift approximations from base domains to product and function space domains. This extension allows analysis of a wide range of programming languages, by abstractly interpreting the domains and operations appearing in the denotational semantics that define them.

In section 3 the denotational approach was generalized even more: we rigorously developed an abstract interpretation framework based on reinterpreting some of the primitive operations appearing in *the metalanguage* used to write denotational language definitions. This yields a framework that is completely independent of any particular programming language.

We now describe some other analysis methods, language properties, and language types. As to methods, we will see that it is sometimes necessary to *instrument* a semantics to make it better suited for analysis, i.e. to modify it so it better models operational aspects relevant to program optimization. (A related approach, not yet as fully developed, is to derive analyses from operational semantics rather than denotational ones.)

Abstract interpretation has its roots in applications to optimizing compilers for imperative languages, so it is not surprising that the early papers by Cousot on semantically based methods were about such programs. A later wave of activity concerned efficient implementation of high-level functional languages. Recent years have witnessed a rapid growth of research in abstractly interpreting logic programming languages, Prolog in particular. Analysis of both functional and logic programming languages will be discussed briefly.

In contrast to the previous section we only give an overview of basic ideas, motivations and a few examples, together with some references to the relevant literature (large — a bibliography from 1986 may be found in [Nielson, 1986c]).

4.1 Approaches to Abstract Interpretation

We now briefly assess what we did in earlier sections, and give some alternative approaches to program analysis by abstract interpretation. Each has some advantages and disadvantages.

4.1.1 The Cousot Approach

This was the first truly semantics-based framework for abstract interpretation, and had many ideas that influenced development of the field, for example abstraction and concretization functions, natural mathematical conditions on them, and the collecting (or static) semantics. The collecting semantics is "sticky", meaning that it works by binding information describing the program's stores to program points. This provided a natural link to the earlier and more informal flow analysis methods used in optimizing compilers, based on constructing and then solving a set of "data flow equations" from the program.

The abstraction and concretization functions are a pair of functions α : C $\rightarrow$ Abs and γ : Abs $\rightarrow$ C between concrete values C and abstract values Abs. Both are required to be complete lattices, and α, γ must satisfy some fairly stringent conditions (γ must be a Galois insertion from Abs into C).

An important new concept was that of a *program interpretation*, abstracting the program's stores. A partial order on interpretations was defined, making it possible to prove rigorously that one interpretation is a correct abstraction of another. This allows program analysis methods to be proven "safe", i.e. to be correct approximations to actual program behaviour.

A limitation is that the Cousot approach only applies to flow chart programs, and has been difficult to extend to, for example, programs with procedures (examples include [Jones, 1982] and [Sharir, 1981]). Another limitation is in its data: as originally formulated, only stores were abstracted, and no systematic way to extend the framework to more general data types was given.

4.1.2 Logical Relations

This solved the problem of extending the Cousots' methods to more general data, using logical relations as defined in [Reynolds, 1974], [Plotkin, 1980]. (In the discussion above on local conditions for safety, the logical relation was $\leq_\beta$ on values.) Safe approximation of composite domains is defined by induction on the form of the domain definitions, leading to a natural sufficient condition for safety of an abstract interpretation that generalizes the Cousots'. In this approach, concretization is not mentioned at all, nor does it seem to be necessary.

Example works using this approach are [Mycroft, 1986], [Jones, 1986] and [Nielson, 1984].

4.1.3 A Method Based on a Metalanguage

The previous method applies only to one language definition at a time. Yet more generality can be obtained by abstractly interpreting the metalanguage used to write the denotational semantics (typically the lambda calculus). This is done in [Nielson, 1984] and subsequent papers, and is summarized in Section 3. A two-level lambda calculus is used to separate those parts of a denotational semantics that are to be approximated from those to remain uninterpreted.

The approach seems to be inherently "non-sticky", as it concerns approximating intermediate values and lacks a means for talking about program points.

4.1.4 Operationally Based Methods

A major purpose of abstract interpretation is its application to efficient program implementation, e.g. in highly optimizing compilers. For application purposes, it thus seems more relevant to use analyses based on a semantics of the language being analyzed, rather than on the metalanguage in which the semantics is written.

But there is a fly in the ointment: many implementation dependent properties relevant to program optimization are simply *not present* in a standard denotational semantics. (This is not at all surprising, if we recall that the original goal of denotational semantics [Stoy, 1977] was to assign the right input-output behaviour to the programs *without* giving implementation details.) Examples include:

- the dependence analyses mentioned in the first section (for neededness analysis, or partial evaluation)
- sequential information about values, e.g. that a variable grows monotonically
- order of parameter evaluation
- time or space usage
- available expressions.

Interesting properties that can be extracted from a denotational semantics include strictness analysis and a (rather weak form of) termination [Burn, 1986], [Abramsky, 1990]. In the approach of [Abramsky, 1990] this is done by focusing on logical relations and then developing "best interpretations" with respect to these: the notion of safety leads to a strictness analysis, whereas the dual notion of liveness leads to a termination of analysis. This is all related to adjunctions between categories and the use of the formula for Kan extensions. However, while a compositional strictness analysis is indeed useful, a compositional termination analysis is not because it has to "give up" for recursion; amending this would entail finding a well-founded order with respect to which the recursive calls do decrease and this is beyond the development of [Abramsky, 1990].

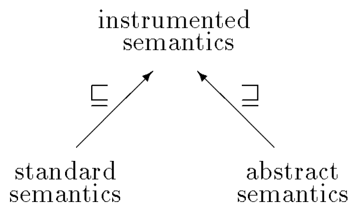
Operational Semantics It would seem obvious to try to extract these properties from an *operational semantics* [Plotkin, 1981], [Kahn, 1987]. However this is easier said than done, for several reasons. One is that there is no clearly agreed standard for what is allowed to appear in an operational semantics, other than that it is usually given by a set of conditional logical inference rules (for instance single-valuedness of an expression evaluation function must be proven explicitly). Another is that operational semantics do not by their nature give all desirable information, e.g. the resource usage or dependency information mentioned above.

It must be said, though, that operational semantics has a large potential as a basis for abstract interpretation, since it more faithfully models actual computational processes, as witnessed by the many unresolved problems concerning full abstraction in denotational semantics. Further, operational semantics is typically restricted to first-order values and so avoids sometimes painful questions involving Scott domains. For instance, there are still several open questions about how to approximate power domains for abstract interpretation of nondeterministic programs.

4.1.5 Instrumented Semantics

Much early work in static program analysis was based on approximating informal models of program execution - and was complex and sometimes wrong. On the other hand, the validity of certain program optimizations and compilation techniques may depend strongly on execution models, e.g. some of the properties just mentioned.

A denotationally based method to obtain information about program execution is to *instrument* the standard semantics, extending it to include additional detail, perhaps operational. The approach can be described by the following diagram, where the left arrow follows since the instrumented semantics extends the standard one, and the right one comes from the requirement of safe approximation. On the other hand all three are obtained by various interpretations of the core semantics, so analysis is still done by abstractly executing source programs.



The collecting semantics illustrates one way to instrument, by collecting the state sets at program points. This is practically significant since many interesting program properties are functions of the sets of states that occur at the program's control points.

More general instrumentation could record a trace or history of the entire computation, properties of the stores or environments, forward or backward value dependencies, sequences of references to variables and much more. This could be used to collect forward dependence information, monotone value growth or perform step counting. The sketched approach puts the program flow analyses used in practice on firmer semantical foundations.

The absence of an arrow between the standard and the abstract semantics is disturbing at first, since correctness is no longer simply a matter of relating abstract values to the concrete ones occurring in computations. However this is inevitable, once the need to incorporate some level of operational detail has been admitted. One must be sure that the extension of the standard semantics properly models the implementation techniques on which optimization and compilation can be based; and this cannot be justified on semantical grounds alone.

On the other hand, an approximate semantics can be proven correct with respect to the instrumented version by exactly the methods of the previous sections.

4.2 Examples of Instrumented Semantics

The possible range of instrumented semantics is enormous, and many variants have already been invented for various optimization purposes. Here we give just a sampling.

4.2.1 Program Run Times

A program time analysis can be done by first extending the standard semantics to record running times, and then to approximate the resulting instrumented semantics. Here we use the denotational framework of section 2.7. The earlier semantics is extended by accumulating, together with the store, the time since execution began.

The time interpretation This is $\mathbf{I}_{time}$ = (Val, Sto; assign, seq, cond, while), defined by

Domains

$$\begin{aligned}
\text{Val} &= \text{Number (the flat cpo)} \\
\text{Sto} &= (\text{Var} \to \text{Val}) \times \text{Number}
\end{aligned}$$

Function definitions

$$\begin{aligned}
\text{assign} &= \lambda(\text{x}, \text{m}_e)\ .\ \lambda(\text{s,t})\ .\ (\text{s}[\text{x} \mapsto \text{m}_e\text{s}], \text{t+1}) \\
\text{seq} &= \lambda(\text{m}_{1c}, \text{m}_{2c})\ .\ \text{m}_{2c} \circ \text{m}_{1c} \\
\text{cond} &= \lambda(\text{m}_e, \text{m}_{1c}, \text{m}_{2c})\ .\ \lambda(\text{s,t}). \\
&\qquad\qquad \text{m}_e\ \text{s} \neq 0 \to \text{m}_{1c}\ (\text{s, t+1}), \text{m}_{2c}\ (\text{s, t+1}) \\
\text{while} &= \lambda(\text{m}_e, \text{m}_c)\ .\ \text{fix}\ \lambda\phi\ .\ \lambda(\text{s,t}). \\
&\qquad\qquad \text{m}_e\ \text{s} \neq 0 \to \phi(\text{m}_c(\text{s,t+1})), (\text{s,t+1})
\end{aligned}$$

This was used as the basis for the approximate time analyses reported in [Rosendahl, 1989].

4.2.2 Execution Traces

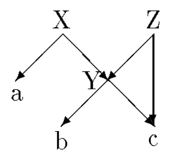
A closely related idea is to instrument a semantics by including full computation histories. This gives in a sense all the raw material that can be used to extract "history-dependent" information, and thus a basis for a very wide range of program analyses. This approach was used in P. Cousot's thesis work, and has since been seen in [Donzeau-Gouge, 1978] and [Nielson, 1982].

In a functional setting, Sestoft has traced sequences of variable definitions and uses (i.e. bindings and references) in order to see which variables can be "globalized", i.e. allowed to reside in global memory instead of the computation stack [Sestoft, 1989]. A similar idea is used in [Bloss and Hudak, 1985] for efficient implementation of lazy functional languages; they trace references to functions' formal parameters to see which ones can be computed using call by value, i.e. prior to function entry.

4.2.3 Store Properties

The sequencing information just mentioned is quite clearly not present in a standard semantics. Another information category very useful for optimization has to do with store properties. A standard semantics for a language with stuctured values often treats them simply as trees. i.e. terms; but in reality memory sharing is used, so new terms are constructed using pointers to old ones rather than a very expensive recopying. For efficient implementation, compile-time analyses must take sharing into account, for example to minimize costs of memory allocation and gargage collection. (Careful proofs of equivalence between the two formulations for a term rewriting language can be seen in [Barendregt et al, 1989].)

In [Jones, 1981a] a simple imperative language with Lisp-like primitives is discussed. The semantics is described operationally, using finite graphs for the store. The following diagram describes a store with X = a :: (b :: c), Y = b :: c and Z = (b :: c) :: c, where Y is shared by X and Z.



This instrumented semantics is modelled in [Jones, 1981a] by approximating such stores by "k-limited graphs", where k is a distance parameter. The idea is that the graph structure is modelled exactly for nodes of distance k or less from a variable. All graph nodes farther than k from a variable name are modelled by nodes of the forms "?c", "?s" or "?", indicating that the omitted portion of the graph is (respectively) possibly cyclic; acyclic but with possibly shared nodes; or acyclic and without sharing.

Stransky has done further work in this direction [Stransky, 1990].

4.3 Analysis of Functional Languages

Motivations for analysing functional languages are partly to counter the time costs involved in implementing powerful programming features such as higher order functions, pattern matching, and lazy evaluation; and partly to reduce their sometimes large and not easily predictable space requirements.

The problem of strictness analysis briefly mentioned in Section 1.1 has reeived much attention, key papers being [Burn, 1986], [Wadler, 1987], and [Hughes, 1990]. Polymorphism, which allows a function's type to be used in several different instantiations, creates new problems in abstract interpretation. Important papers include [Abramsky, 1986] and [Hughes, 1990].

Approximating functions by functions on abstract values A natural and common approach is to let an abstraction of a function be a function on abstract values. One example is Section 3's general framework using logical relations and based on the lambda calculus as a metalanguage.

The first higher-order strictness analysis was the elegant method of [Burn, 1986]. In this work the domains of a function being analyzed for strictness are modeled by abstract domains of exactly the same structure, but with $\{\bot, \top\}$ in place of the basis domains. Strictness information is in essence obtained by computing with these abstracted higher-order functions. However, a fixpoint iteration is needed, since the abstractions sacrfice the program's determinacy (cf. the end of Section 1.1).

While suitable for many problems concerning functional and other languages, abstracting functions by functions is not always enough for the program analyses used in practice. An example where this simply does not give enough information is *constant propagation* in a functional language. As before, the goal is to determine whether or not one of the arguments of a program function is always called with the same value and, if so, to find that value at analysis time.

For this it is not enough to know that if a function f is called with argument x, it will return value $x + 1$. It is also essential to know *which values f can be called with* during program execution, since a compiler can exploit knowledge about constant arguments to generate better target code.

For another example, the higher-order strictness analysis in [Burn, 1986] has turned out to be unacceptably slow in practice, even for rather small programs. The reason is that the abstract interpretation involves computing fixpoints of higher-order functions. Abstract domains for the function being analyzed have the same structure as the originals, but with $\{\perp, \top\}$ for all basis domains. This implies that analysis of even a small function such as "fold right" can lead to a combinatorial explosion in the size of the abstract domains involved, requiring subtle techniques to be able to compute the desired strictness information and avoid having to traverse the entire abstract value space.

4.3.1 First Order Minimal Function Graphs

The *minimal function graph* of a program function was defined in [Jones, 1986] to be the smallest set of pairs (argument, function value) suffcicient to carry out program execution on given input data. For example, consider the function defined by the following program:

$$\begin{array}{rcl} f(X) & = & \mathbf{if}\, X = 1\, \mathbf{then}\, 1 \\ & & \mathbf{else\, if}\, X even\, \mathbf{then}\, f(X/2) \\ & & \mathbf{else}\, f(3 * X + 1) \end{array}$$

Its minimal function graph for program input X = 3 is

$$\{(3,1),(10,1),(5,1),(16,1),(8,1),(4,1),(2,1),(1,1)\}$$

The minimal function graph semantics maps a programmer-defined function to something more detailed than the argument-to-result function traditionally used in a standard semantics, and is a form of instrumented semantics. In [Jones, 1986] it is shown how the "constant propagation" analysis may be done by approximating this semantics, and the idea of proving correctness by semihomomorphic mappings between various interpretations of a denotational semantics is explained.

Earlier methods to approximate programs containing function calls were described in [Cousot, 1977c], [Sharir, 1981] and [Jones, 1982].

4.3.2 Higher Order Functions

Higher order functions as well as first order ones may be approximated using logical relations. Examples include the analyses of Section 3.2.3 and [Burn, 1986], but such methods cannot closely describe the way functions are used during execution.

For example, if the value of exp in an application $exp(exp')$ is modelled by a function from abstract values to abstract values, this does not contain enough information to see just *which programmer-defined functions* may be called at run time; and such information may be essential for efficient compilation.

This leads to more operational approaches to abstractly interpreting programs containing higher order functions. An early step in this direction was the rather complex [Jones, 1981b], and more recent and applications-motivated papers include [Sestoft, 1989,Shivers, 1991].

Closure analysis This method from can be described as an operationally oriented semantics-based method. Programs are assumed given as systems of equations in the now popular named combinator style, with all functions curried, and function definitions of the form

$$fX_1 \ldots X_n \quad = \quad \text{expression containing } X_1 ... X_n \text{ and function names}$$

In this language, a value which is a function is obtained by an incomplete function application. Operationally such a value is a so-called *closure* of form $< fv_1 \ldots v_i >$ where $v_1 \ldots v_i$ are the values of f's first i arguments.

The paper [Sestoft, 1989] contains algorithms for a closure analysis, yielding for example information that in a particular application $exp(exp')$, the operator exp can evaluate to an f-closure with one evaluated argument, or to a g-closure with two evaluated arguments.

Instead of approximating a function by a function, each programmer-defined function f is described by a *global function description table* containing descriptions of all arguments with which it can be called (just as in the minimal function graphs discussed earlier).

An approximation to a value which is a function is thus represented by an approximate closure of the form $< fa_1 \ldots a_i >$ where the a_i approximate the v_i.

But then how is v_i itself approximated? (There seems to be a risk of infinite regression.) Supposing v_i can be a closure $< gw_1 \ldots w_j >$, we can simply approximate it by the pair $< g, j >$. The reason this works is that, when needed, a more precise description of g's arguments can be obtained from g's entry in the global function description table.

Analysis starts with a single global function description table entry describing the program's initial call, and abstract interpretation continues until this table stabilizes, i.e. reaches its fixpoint. Termination is guaranteed because there are only finitely many possible closure descriptions.

Closure analysis describes functions globally rather than locally, and so appears to be less precise in principle than approximating functional values by mathematical functions. This is substantiated by complexity results that show the analyses of [Burn, 1986] to have a worst-case lower bound of exponential time, whereas closure analysis works in time bounded by a low-degree polynomial.

The techniques developed in [Sestoft, 1989,Shivers, 1991] have shown themselves useful for a variety of practical flow analysis problems involving higher order functions, for example in efficient implementation of lazy evaluation [Sestoft and Argo, 1989] and partial evaluation [Bondorf, 1991]. Similar ideas are used in [Jones, 1993b] to provide an operationally oriented denotational minimal function graph semantics for higher order programs. One application is to prove the safety of Sestoft's algorithms.

4.3.3 Backwards Analysis and Contexts

As observed in Section 2.5.3, backwards analysis of an imperative program amounts to finding the weakest precondition on stores sufficient to guarantee that a certain postcondition will hold after command execution. For functional programs, an analogous concept to postcondition is that of the *context* of a value, which describes the way the value will be used in the remainder of the computation.

Clearly the usage of the result of a function will affect the usage of its arguments to the function will be used. For an extreme example, if the result of function call $f(e_1, \ldots, e_n)$ is not needed, then the arguments $e_1, \ldots, e_n$ will not be needed either.

The example is not absurd; consider the following abstract program, using pattern matching and a list notation where nil = [] is the empty list, : is the concatenation operator, $[\mathrm{a}_1, \ldots, \mathrm{a}_n]$ abbreviates $\mathrm{a}_1 : \ldots : \mathrm{a}_n$: nil.

$$\begin{array}{lcl} \mathit{length}\,([]) & = & 0 \\ \mathit{length}(\mathrm{Z{:}Zs}) & = & 1 + \mathit{length}(\mathrm{Zs}) \\ \mathit{f}(\mathrm{X}) & = & \textbf{if } \mathit{test}(\mathrm{X}) \textbf{ then } [] \textbf{ else } g(\mathrm{X}) : \mathit{f}(\mathrm{X\text{-}1}) \end{array}$$

When evaluating a call $\mathit{length}(f(exp))$, the values of $g(\ldots)$ are clearly irrelevant to the length of $f(exp)$, and g need not be called at all. (This can be used to optimize code in a lazy language.) For another example, if $f(n, x) = x^n$ and a call $f(e_1, e_2)$ appears in a context where its value is an even number, one can conclude that e_1 is positive and e_2 is even.

Context information thus propagates backwards through the program: from the context of an enclosing expression to the contexts of its subexpressions, and from the context of a called function to the contexts of its parameters in a call.

Some uses of backwards functional analyses

Strictness analysis identifies arguments in a lazy or call by name language for which the more efficient call by value evaluation may be used without changing semantics. Both forwards and backwards algorithms exist, but backwards methods seem to be faster. Early work on backwards methods includes [Hughes, 1985] and [Hughes, 1987], later simplified for the case of domain projections in [Wadler, 1987] and [Hughes, 1990].

Storage reclamation. Methods are develped in [Jensen, 1991] to recognize when a memory cell has been used for the last time, so it may safely be freed for later use. An application is substantially to reduce the number of garbage collections.

Partial evaluation automatially transforms general programs into versions specialized to partially known inputs. A specialized program is usually faster that the source it was derived from, but often contains redundant data structures or unused values (specializations of general-purpose data in the source). Backwards analysis is used for "arity raising", which improves programs by removing unnecessary data and computation [Romanenko, 1990].

Information flow While the analogy to the earlier backwards analyses is clear, the technical details are different and rather more complicated. A bottleneck is that, while functions may have many arguments, they produce only one result. Thus one cannot simply invert the "next" relation to describe program running in the reverse direction.

Given a function definition

$$f X_1 \ldots X_n \quad = \quad \text{expression containing } X_1 \ldots X_n \text{ and function names}$$

one can associate a context transformer $f^i : Context \rightarrow Context$ with each argument X_i. The idea is that if f is called with a result context C, then $f^i(C)$ will be the context for f's ith argument.

In effect this is an independent attribute formulation of f's input-output relation, necessarily losing intervariable interactions. Unfortunately it means that backwards analyses cannot in principle exactly describe the program's computations, as is the case with forward analyses.

What is a context, semantically? The intuition "the rest of the computation" can be expressed by the *current continuation*, since since a continuation is a function taking the current expression value into the program's computational future. This approach was taken in [Hughes, 1987], but is technically rather complex since it entails that a context is an abstraction of a set of continuations—tricky to handle since continuations are higher-order functions.

In the later [Wadler, 1987] and [Hughes, 1990], the concept of context is restricted to properties given by *domain projections*, typically specifing which parts of a value might later be used. A language for finite descriptions of projections and their manipulation was developed, flow equations were derived from the program to be analyzed, and their least fixpoint solution gives the desired information.

Some example contexts that have shown themselves useful for the efficient implementation of lazy functional programs include:

- ABSENT: the value is not needed for further computation

- ID: the entire value may be needed
- HEAD: the value is a pair, and its first component may be needed (but the second will not be)
- SPINE: the value is a list, and all its top-level cells may be needed (the *length* function demads a SPINE context of its argument)

4.4 Complex Abstract Values

Finding finite approximate descriptions of infinite sets of values is an essential task in abstract interpretation. We mentioned in Section 2.5 that abstracting stores or environments amounts to finding finite descriptions of relations among the various program variables. This was straightforward in the even-odd example given earlier, e.g. "odd" represented {1,3,5,...}, etc., and operations on numbers were easily modeled on this finite domain of abstract values. Analysis problems requiring more sophisticated methods include

- functions as values (especially higher order functions)
- mutual relationships among variable values
- describing structured data, e.g. nested lists and trees

4.4.1 Functions as values

Some approaches were described above (approximation by functions on abstract values, and closure analysis), and several more have been studied.

4.4.2 Relations on n-tuples of numbers

In this special case there is a well-developed theory: linear algebra, in particular systems of linear inequalities. [Cousot, 1978] describes a way to discover linear relationships among the variables in a Pascal-like program. Such relations may be systematically discovered and exploited for, for example, efficient compilation of array operations. Related work, involving the inference of systems of modulus equations, has been applied to pipelining and other techniques for utilizing parallelism [Granger, 1991, Mercouroff, 1991]. This work has been further developed into a system for automatic analysis of Pascal programs.

4.4.3 Grammars

The analysis of programs manipulating structured data, e.g. lists as in Lisp, ML, etc, requires methods to approximate the infinite sets of values that variables may take on during program runs. There is also a well-developed theory and practice for approximating such infinite sets, involving *regular grammars* or their equivalent, *regular expressions*.

An Example Grammar Construction Consider the following abstract program, using the list notation of Section 4.3.3.

f(N)	=	first(N, sequence(nil))
first(nil, Xs)	=	nil
first(M : Ms, X : Xs)	=	Ms : first(Ms, Xs)
sequence(Y)	=	Y : sequence(1 : Y)

We assume an initial call of form f(N) where the input variable N ranges over all lists of 1's, and further that the language is lazy. Conceptually, call "sequence(nil)" generates the infinite list [[],[1],[1,1],[1,1,1],...]. The possible results of the program are all of its finite prefixes:

$$\text{Output} = \{[], [[],[1]], [[],[1],[1,1]], [[],[1],[1,1],[1,1,1]], \ldots\}$$

The method of [Jones, 1987a] constructs form this program a tree grammar G containing (after some simplification) the following productions. They describe the terms which are the program's possible output values:

N	::=	nil | 1 : N	Program input = f argument
f_{result}	::=	first_{result}	
first_{result}	:=	nil | Ms : first_{result}	
M	::=	1	
Ms	::=	N	
X	::=	Y	
Xs	::=	sequence_{result}	
Y	::=	nil | 1 : Y	
sequence_{result}	::=	Y : sequence_{result}	

With f_{result} as initial nonterminal, G generates all possible lists, each of whose elements is a list of 1's. More generally, by this approach, an abstract value in Abs is a tree grammar, and the concretization function maps the tree grammar and one of its nonterinal symbols A into the set of terms that A generates.

Safety The natural definition of *safe program approximation* is that the grammar generates all possible runtime values computed by the program (and usually a proper superset). By the grammar above, nonterminal f_{result} clearly generates all terms in Output — and so is a safe approximation to the actual program behaviour.

It is not a perfect description, since f_{result} generates all possible lists of lists of 1's, regardless of order.

Nonexistence of an abstraction function α For this analysis an abstract value is a large object: a grammar G, and the concretization function γ maps G's nonterminals into term sets which are supersets of the value sets that variables range over in actual computations. It is natural to ask: what is the corresponding abstraction function α?

In this case, there is no unique natural α, for a mathematical reason. The point is that regular tree grammars as illustrated above generate only *regular sets of terms*, a class of sets with well-known properties. On the other hand, the program above, and many more, generate nonregular sets of values (Output is easily proven nonregular). It is well known that for any nonregular set S of terms, there is no "best" regular superset of S. In general, increasing the number of nonterminals will give better and better "fits", i.e. smaller supersets, but a perfect fit to a nonregular set is (by definition) impossible.

References The papers [Reynolds, 1969] and [Jones, 1981a,Jones, 1987a] contain methods to construct, given a program involving structured values, a regular tree grammar describing it. Essentially similar techniques, although formalized in terms of tables rather than grammars, have been applied to the lambda calculus [Jones, 1981b], interprocedural imperative program analysis [Jones, 1982], a language suitable as an intermediate language for ML [Deutsch, 1990], and the Prolog Language [Heintze, 1992].

4.5 Abstract Interpretation of Logic Programs

Given the framework already developed, we concentrate on the factors that make logic program analysis different from those seen earlier. Further, we concentrate on Prolog, in which a program is a sequence of *clauses*, each of the form "head $\leftarrow$ body":

$$\mathrm{h}(\mathrm{t}_1,\ldots,\mathrm{t}_m) \leftarrow \mathrm{b}_1(\mathrm{t}'_1,\ldots,\mathrm{t}'_n) \wedge \ldots \wedge \mathrm{b}_k(\mathrm{t}''_1,\ldots,\mathrm{t}''_p).$$

where h, $\mathrm{b}_1,\ldots,\mathrm{b}_k$ are *predicate names* and the t_i are *terms* built up from *constructors* and *variables*. Variable names traditionally start with capital letters, e.g. X. Constructors are as in functional languages (e.g. ":" and "[]"), but runtime values are rather different, as they may contain free or "uninstantiated" variables. Each $\mathrm{b}_i(\mathrm{t}_1,\ldots)$ is called a *goal*.

An example program, for appending two lists:

append(Xs, [], Xs) $\leftarrow$.
append(X:Xs,Ys,X:Zs) $\leftarrow$ append(Xs,Ys,Zs).

4.5.1 Semantics of Prolog Programs

Prolog can be viewed either as a pure logical theory (so a program is a set of "Horn clauses" with certain logical consequences), or as an operationally oriented programming language. When used operationally, programs may contain features with no interpretation in mathematical logic, to improve efficiency or facilitate communication with other programs. Examples: input/output operations, tests as to whether a variable is currently instantiated, and operations to add new clauses to the program currently running, or to retract existing clauses.

A *ground* term is one containing no variables. The *bottom-up* or logical interpretation of "append" is the smallest 3-ary relation on ground terms which satisfies the implications in the program. It thus contains append(1:[],2:[],1:2:[]) and append(1:2:[],3:4:[],1:2:3:4:[]), among others.

Top-down interpretation is used for Prolog program execution. Computation begins with a query, which is a "body" as described above. The result is a finite or infinite sequence of *answer substitutions*, each of which binds some of the free variables in the query. In a top-down semantics, the basic object of discourse is not a store or an environment, but a substitution that maps variables to new terms — which may in turn contain uninstantiated variables, i.e. be nonground.

The result of running the program above with an initial query append(1:2:[],3:4:[],Ws) would be the one-element answer sequence

$$[\text{Ws} \mapsto 1{:}2{:}3{:}4{:}[\,]]$$

while the result of running with query append(Us,Vs,1:2:[]) would be the sequence of three answers

$$\begin{array}{l}[\text{Us} \mapsto 1{:}2{:}[\,],\ \text{Vs} \mapsto [\,]]\\ [\text{Us} \mapsto 1{:}[\,],\ \text{Vs} \mapsto 2{:}[\,]]\\ [\text{Us} \mapsto [\,],\ \text{Vs} \mapsto 1{:}2{:}[\,]]\end{array}$$

and the result of running with query append(1:2:[],Vs,Ws) would be an answer containing an uninstantiated variable:

$$[\text{Vs} \mapsto \text{Ts},\ \text{Ws} \mapsto 1{:}2{:}\text{Ts}]$$

4.5.2 Special Features of Logic Programs

The possibility of more than one answer substitution is due to the *backtracking* strategy used by Prolog to find all possible ways to satisfy the query. These are found by satisfying the individual goals $q_i(\ldots)$ left to right,*unifying* each with all possible clause left sides in the order they appear in the program. Once unification with a clause head has been done, its body is then satisfied in turn (trivially true if it is empty). This procedure is often described as a depth-first left-to-right search search of the "SLD-tree".

Bindings made when satisfying q_i are used when satisfying q_j for $j > i$, so in a certain sense the current substitution behaves like an updatable store. A difference is that it is "write-once" in that old bindings may not be changed. However changes may be made by instantiating free variables.

4.5.3 Types of Analyses

All this makes program analysis rather complex, and the spectrum seen in the literature is quite broad. Many but not all analyses concentrate on "pure" Prolog subsets without nonlogical features. Some are bottom-up, and others are top-down.

4.5.4 Needs for Analysis

A first motivation for analysis is to optimize memory use — Prolog is notorious for using large amounts of memory. One reason is that due to backtracking the information associated with a predicate call cannot be popped when the call has been satisfied, but must be preserved for possible future use, in case backtracking should cause control the call to be performed again. Considerable research on "intelligent backtracking" is being done to alleviate this problem, and involves various abstract interpretations of possible program behaviour.

Another motivation is speed. Unification of two terms containing variables is a fundamental operation, and one that is rather slower than assignment in imperative languages, or matching as used in functional languages. One reason is that unification involves *two-way bindings*: variables in either term may be bound to parts of the other term (or transitively even to parts of the same term). Another is the need for the *occur check*: to check that a variable never gets bound to a term containing itself (although sometimes convenient for computing, programs containing such "circular" terms have no natural logical interpretation).

4.5.5 Examples of Analysis

Analyses to speed up unification The following are useful to recognize when special forms of unification can be used such as assignment or one-way matching:

Mode analysis determines for a particular goal those of its free variables which will be unbound whenever a goal is called, and which will be bound as a result of the call [Mellish, 1987].

Groundness anaysis discovers which variables will always be bound to a ground (variable-free) term when a given program point is reached. For example "append", when called with a query with ground Xs and Ys, yield a substitution with ground Zs; when called with bround Zs, all answers will have ground Xs and Ys; and when called with only Xs ground, the answers will never be ground.

Groundness analysis is more subtle than it appears due to possible aliasing and shared substructures, since binding one variable to a ground term may affect variables appearing in another part of the program being analyzed. Simple groundness and sharing analyses are described in [Jones, 1987b]. A more elegant method using propositional formulas built from $\wedge$ and $\Leftrightarrow$ was introduced in [Marriott, 1987] and compared with other methods in [Cortesi, 1991].

Safely avoiding the occur check

Circularity analysis is a related and more subtle problem, the goal being to discover which unifications may safely be performed without doing the time-consuming "occur check". The first paper on this was by Plaisted [Plaisted, 1984], with a very complex and hard to follow method. A more semantics-based method was presented in [Søndergaard, 1986].

Other analyses The "difference list" transformation can speed programs up by nonlinear factors, and can be applied systematically; but it can also change program semantics if used indiscriminately. Analyses to determine when the transformation may be safely applied are described in [Marriott, 1988].

Other uses include binding time analysis for offline partial evaluation, and deciding when certain optimizing transformations can be applied. One example is deforestation [Wadler, 1988].

4.5.6 Methods of Analysis

Analysis methods can roughly be divided into the pragmatically oriented, including [Bruynooghe, 1991], [Mellish, 1987], and [Nilsson, 1991]; and the semantically oriented, including [Cortesi, 1991], [Debray, 1986], [Jones, 1987b], and [Marriott, 1993].

A natural analogue to the accumulating semantics seen earlier was used in [Jones, 1987b] and a number of later papers, and presumes given a sequence of clauses and a single query. It is a "sticky" semantics in which the program points are the positions just before each clause goal or the query, and at the end of each clause and the query. With each such point is accumulated the set of all substitutions that can obtain there during computations on the given query (so those for the query end describe the answer substitutions).

Approximation of substitutions and unification. These are nontrivial problems for several reasons. One is renaming: each clause is (implicitly) universally quantified over all variables appearing in it, so unification of a predicate call with a clause head requires renaming to avoid name clashes, and the same variable may appear in many "incarnations" during a single program execution. Many approximations merge information about all incarnations into a single abstraction, but this is not safe for all analyses.

Aliasing is also a problem, since variables may be bound to one another so binding one will change the bindings of all that are aliased with it. Terms containing free variables have the same problem: binding one variable will change the values of all terms containing it.

Finally, unification binds variables to structured terms, so approximations that do not disregard structure entirely have some work to do to obtain finite descriptions. A recent example is [Heintze, 1992].

[Marriott, 1993] is interesting in two respects: it uses the metalanguage approach described in this work, in Section 3; and it uses sets of constraints instead of substitutions, reducing some of the technical problems just mentioned.

5 Glossary

abstraction function: Usually a function from values or sets of values to abstract values such as EVEN, ODD. See adjoined pair.

accumulating semantics: A semantics that models the set of values that a standard semantics (or instrumented semantics) may produce. The functionality of commands might be $\mathcal{P}(\text{Sto}) \rightarrow \mathcal{P}(\text{Sto})$, and the program description might be $\text{Pla} \rightarrow \mathcal{P}(\text{Sto})$, where Pla is the domain of program points (or places).

adjoined pair: A pair of functions $(\alpha{:}D{\rightarrow}E,\gamma{:}E{\rightarrow}D)$ that satisfies $\alpha(d)\sqsubseteq e$ if and only if $d\sqsubseteq\gamma(e)$. The first component is often called an *abstraction function* (or a *lower adjoint*) and the second component is called a *concretization function* (or an *upper adjoint*).

backward: Used for an analysis where the program is analysed in the opposite direction of the flow of control, an example being liveness analysis (see glossary). In Section 3 this is formalized by interpreting $\underline{\rightarrow}$ as $\leftarrow$ where $D{\leftarrow}E$ means $E{\rightarrow}D$.

collecting semantics: Has been used to mean *sticky semantics* as well as *lifted* (or accumulating) *semantics*, so some confusion as to its exact meaning has arisen in the literature.

concretization function: Usually a function from abstract values such as EVEN, ODD to sets of concrete values. See adjoined pair.

context: A description of how a computed value will be used in the remainder of the computation. Used in backwards analysis of functional programs.

correctness: Given a relation of theoretical or implementational importance, correctness amounts to showing that the properties obtained by abstract interpretation always have this relation to the standard semantics.

core semantics: See factored semantics.

duality principle: The principle of lattice theory saying that by changing the partial order from $\sqsubseteq$ to $\sqsupseteq$ one should also change least fixed points to greatest fixed points and least upper bounds to greatest lower bounds and that then the same information results.

factored semantics: The division of a denotational semantics into two parts: a core, assigning terms to every language construct, but with some details left unspecified; and an interpretation, giving the meanings of the omitted parts. Often used to compare the correctness of one abstract interpretation with respect to another abstract interpretation.

first-order: Used for an analysis where the properties directly describe actual values. An example is *detection of signs* where the property '+' describes the values 1, 2, 3, etc.

forward: Used for an analysis where the program is analysed in the same direction as the flow of control. In Section 3 this is formalized by interpreting $\underline{\rightarrow}$ as $\rightarrow$.

independent attribute method: Used for an analysis where the components of a tuple are described individually, ignoring relationships among components. In Section 3 this is formalized by interpreting $\underline{\times}$ as $\times$.

induced property transformer: An analysis that is obtained from a standard semantics or another analysis in a certain way that is guaranteed to produce an optimal analysis over a given selection of properties.

instrumented semantics: A version of the standard semantics where more operational detail is included. In general an instrumented semantics constrains the implementation of a language as defined by its standard semantics.

interpretation: See factored semantics.

lax functor: A modification of the categorical notion of functor in that certain equalities are replaced by inequalities.

lifted semantics: Another term for the *accumulating semantics*.

live variable: A variable whose variable may be used later in the current computation.

logical relation: A relation constructed by induction on a type structure in a certain 'natural' way.

minimal function graph: An interpretation that associates to each user-defined function in a program a subset of $S^* \times S$ that indicates those argument/result pairs which are actually involved in computing outputs for a given input to a program.

relational method: Used for an analysis where the interrelations among components of a tuple are described, e.g. $X + Y < 110$. In Section 3 this is formalized by interpreting $\underline{\times}$ as $\otimes$ (tensor product).

representation transformation: A function that maps values or properties to properties, e.g. an abstraction function.

safety: Essentially the same as correctness, but emphasizing that the results of an analysis may be used for program transformation without changing semantics. Often used when the correctness of one abstract interpretation is established with respect to another abstract interpretation.

second-order: Used for an analysis where the properties do not directly describe actual values but rather some aspects of their use. An example is *liveness* where the property *live* does not describe any value but rather that the value might be used in the future computations.

standard semantics: A semantics where as few implementation considerations as possible are incorporated. The functionality of commands might be Sto$\rightarrow$Sto.

static semantics: Has been used to mean *sticky lifted semantics* but this usage conflicts with the distinction between static and dynamic semantics.

sticky semantics: Used for a semantics which binds program points to various information ("sticky" in the sense of flypaper). In a sticky semantics a command might be of functionality Sto$\rightarrow$Pla$\rightarrow\mathcal{P}$(Sto), where Pla is the domain of program points (or places).

strict function: $f : V_1 \times \ldots V_n \rightarrow W$ is strict in its ith argument if $f(v_1, \ldots, v_{i-1}, \perp, v_{i+1}, \ldots, v_n) = \perp$ for all $v_i \in V_i$.

tensor product: An operation $\otimes$ on algebraic lattices that may be used to formalize the notion of *relational method*. When formulated in the categorical framework, as is natural when recursive types are to be considered, the concept of *lax functor* is necessary.

References

[Abramsky, 1986] S.Abramsky: Strictness analysis and polymorphic invariance, *Programs as Data Objects*, 1–23, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*. (Springer-Verlag, 1985).

[Abramsky, 1987] *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin, eds., (Ellis Horwood, 1987).

[Abramsky, 1990] S.Abramsky: *Abstract interpretation, logical relations and Kan extensions*, *Journal of logic and computation*, **1**(1),1990.

[Aho, Sethi and Ullman, 1986] A.V.Aho, R.Sethi, J.D.Ullman: *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, 1986).

[Ammann, 1994] Jürgen Ammann: Elemente der Projektions-Analyse für Funktionen höherer Ordnung, M.Sc.-thesis, University of Kiel, forthcoming.

[Bandelt, 1980] H-J.Bandelt: The tensor product of continuous lattices, *Mathematische Zeitschrift* **172** (1980) 89–96.

[Barendregt et al, 1989] Term graph rewriting. In J. W. de Bakker, A. J. Nijman, P. C. Treleaven (eds.): *PARLE—Parallel Architectures and Languages Europe* (Lecture Notes in Computer Science, vol. 259), (Springer-Verlag, 1989).

[Bloss and Hudak, 1985] P. Hudak, A. Bloss: The Aggregate Update Problem in Functional Programming, *Proc. 12th ACM Symposium on Principles of Programming Languages* 300–314 (1985).

[Bondorf, 1991] A. Bondorf: Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[Bruynooghe, 1991] M. Bruynooghe: A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal of Logic Programming*, **10**, 91–124, 1991.

[Burn, 1986] G.L.Burn, C.Hankin, S.Abramsky: Strictness Analysis for Higher-Order Functions, *Science of Computer Programming* **7** (1986) 249–278.

[Cortesi, 1991] A.Cortesi, G.File, W.Winsborough: *Prop* Revisited: Propositional Formulas as Abstract Domain for Groundness Analysis. *Proceedings 6th Annual Symposium on Logic in Computer Science*, 322-327 (1991).

[Cousot, 1977a] P.Cousot, R.Cousot: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Proc. 4th ACM Symposium on Principles of Programming Languages* (1977) 238–252.

[Cousot, 1977b] P.Cousot, R.Cousot: Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations, *Proc. ACM Symposium on Artificial Intelligence and Programming Languages* (1977) 1–12.

[Cousot, 1977c] P.Cousot, R.Cousot: Static Determination of Dynamic Properties of Recursive Procedures, *IFIP Working Conference on Programming Concepts* (North-Holland, 1977) 237–277.

[Cousot, 1978] P.Cousot, N.Halbwachs: Automatic Discovery of Linear Restraints Among Variables of a Program, *Proc. 5th ACM Symposium on Principles of Programming Languages* (1978) 84–97.

[Cousot, 1979] P.Cousot, R.Cousot: Systematic Design of Program Analysis Frameworks, *Proc. 6th ACM Symposium on Principles of Programming Languages* (1979) 269–282.

[Curien, 1986] P.-L.Curien: *Categorical Combinators, Sequential Algorithms and Functional Programming* (Wiley, 1986).

[Darlington, 1977] R.M. Burstall and J. Darlington: A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Debray, 1986] S. Debray: Dataflow Analysis of Logic Programs, Report, SUNY at Stony Brook, New York (1986).

[Deutsch, 1990] A. Deutsch: On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher Order Functional Specifications, in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages,* 157–169, (ACM Press, 1990).

[Dijkstra, 1976] E.W. Dijkstra: *A Discipline of Programming* (Prentice-Hall, 1976).

[Donzeau-Gouge, 1978] V. Donzeau-Gouge: Utilisation de la Sémantique dénotationelle pour l'étude d'interpretations non-standard, INRIA rapport 273 (1978).

[Dybjer, 1987] P.Dybjer: Inverse Image Analysis, *Proc. ICALP 87*, Lecture Notes in Computer Science **267** (Springer-Verlag, 1987) 21–30.

[Fosdick, 1976] L.D. Fosdick, L. Osterweil: Data Flow Analysis in Software Reliability. *Computing Surveys*, **8**, 305-330 (1976).

[Foster, 1987] M. Foster: Software Validation using Abstract Interpretation, in *Abstract Interpretation of Declarative Languages,* edited by S. Abramsky and C. Hankin, 32–44, Ellis Horwood, Chichester, England, 1987.

[Gordon, 1979] M.J.C.Gordon: *The Denotational Description of Programming Languages: An Introduction* (Springer-Verlag, 1979).

[Granger, 1991] P. Granger: Static Analysis of Linear Congruence Equalities among Variables of a Program, in *Proceedings of the Fourth International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '91), Lecture Notes in Computer Science 493,* 169–192, (Springer-Verlag, 1991).

[Hecht, 1977] M.S. Hecht: *Flow Analysis of Computer Programs*, (Elsevier North-Holland 1977).

[Heintze, 1992] N.Heintze: Set Based Program Analysis, Carnegie-Mellon Ph.D. thesis, 1992.

[Hudak, 1991] P.Hudak, J.Young: Collecting interpretation of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, April 1991.

[Hughes, 1985] J. Hughes: Strictness Detection in Non-flat Domains, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science.* (Springer-Verlag, 1985).

[Hughes, 1987] John Hughes: Analyzing Strictness by Abstract Interpretation of Continuations, in *Abstract Interpretation of Declarative Languages,* edited by S. Abramsky and C. Hankin, 63–102, Ellis Horwood, Chichester, England, 1987.

[Hughes, 1988] John Hughes: Backward Analysis of Functional Programs, *Partial evaluation and mixed computation*, 187–208, North-Holland, 1988.

[Hughes, 1990] R.J.M. Hughes, J. Launchbury: Projections for Polymorphic Strictness Analysis, *Mathematical Structures in Computer Science* (1990).

[Jensen, 1991] Thomas P. Jensen:. Strictness analysis in logical form. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, 1991.

[Jensen, 1991] Thomas P. Jensen, T.Æ. Mogensen: A Backwards Analysis for Compile Time Garbage Collection. In *Proceedings of the European Symposium on Programming*, LNCS **432** (Springer-Verlag 1991).

[Jones, 1986] N.D.Jones, A.Mycroft: Dataflow of Applicative Programs Using Minimal Function Graphs, *Proc.13th ACM Symposium on Principles of Programming Languages* (1986) 296–306.

[Jones, 1981a] S.S. Muchnick, N.D. Jones: Flow Analysis and Optimization of Lisp-like Structures, in *Program Flow Analysis*, eds. S.S. Muchnick, N.D. Jones (Prentice-Hall 1981).

[Jones, 1981b] N.D. Jones: Flow Analysis of Lambda Expressions, *ICALP 1981, Lecture Notes in Computer Science.* Springer-Verlag (1981).

[Jones, 1982] N.D. Jones, S.S. Muchnick: A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures,in *Proceedings of the 6th ACM Symposium on Principles of Programming Languages,* 66–74 (1982).

[Jones, 1987a] Neil D. Jones: Flow Analysis of Lazy Higher-Order Functional Programs, in *Abstract Interpretation of Declarative Languages,* edited by S. Abramsky and C. Hankin, 103–122, Ellis Horwood, Chichester, England, 1987.

[Jones, 1987b] Neil D. Jones, Harald Søndergaard: A Semantics-Based Framework for the Abstract Interpretation of Prolog, in *Abstract Interpretation of Declarative Languages,* edited by S. Abramsky and C. Hankin, 123–142, Ellis Horwood, Chichester, England, 1987.

[Jones, 1989] Neil D. Jones, Peter Sestoft, Harald Søndergaard: Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation, *Lisp and Symbolic Computation* 2,1 (1989) 9–50.

[Jones, 1993a] N.D. Jones, C. Gomard, P. Sestoft: Partial Evaluation and Automatic Program Generation. Prentice Hall International, 1993.

[Jones, 1993b] N.D. Jones, M. Rosendahl: Higher Order Minimal Function Graphs, DIKU report.

[Kahn, 1987] G. Kahn: Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247)*, 22–39 (Springer-Verlag, 1987).

[Kam, 1976] J.B. Kam, J.D. Ullman: Monotone Data Flow Analysis Frameworks, *Acta Informatica* **7**, 305–317 (1977).

[Kennedy, 1981] K.Kennedy: A Survey of Compiler Optimization, in *Program Flow Analysis*, eds. S.S. Muchnick, N.D. Jones (Prentice-Hall 1981).

[MacLane, 1971] S.MacLane: *Categories for the Working Mathematician*, (Springer-Verlag, 1971).

[Marriott, 1987] K. Marriott, H. Søndergaard: Notes for a Tutorial on Abstract Interpretation of Prolog Programs, North American Conference on Logic Programming (1987).

[Marriott, 1988] K. Marriott, H. Søndergaard: Prolog Program Transformation by the Introduction of Difference-lists, accepted by *ACM Transactions on Programming Languages and Systems* (ACM, 1988).

[Marriott, 1993] K. Marriott, H. Søndergaard, N.D. Jones: Denotational Abstract Interpretation of Prolog Programs, in *Proceedings of International Computer Science Conference*, 206–213 (IEEE Computer Society, 1993?).

[Masdupuy, 1991] F. Masdupuy: Using Abstract Interpretation to Detect Array Data Dependencies, in *Proceedings of the International Symposium on Supercomputing*, Fukuoka, 19-27 (1991).

[Mellish, 1987] Abstract Interpretation of Prolog Programs, in *Abstract Interpretation of Declarative Languages,* edited by S.

Abramsky and C. Hankin, 181–198 (Ellis Horwood, 1987).

[Mercouroff, 1991] N. Mercouroff: An Algorithm for Analysing Communicating Processes, in *The Mathematical Foundations of Programming Semantics Conference* (1991).

[Muchnick, 1981] S.S. Muchnick, N.D. Jones: *Program Flow Analysis* (Prentice-Hall 1981).

[Mycroft, 1981] A. Mycroft: *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, 1981. Also report CST-15-81.

[Mycroft, 1983] A.Mycroft, F.Nielson: Strong Abstract Interpretation using Power Domains, *Proc. ICALP 1983*, Lecture Notes in Computer Science **154** (Springer-Verlag, 1983) 536–547.

[Mycroft, 1986] Alan Mycroft, Neil D. Jones: A relational framework for abstract interpretation. In H. Ganzinger and N. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Mycroft, 1987] A.Mycroft: A Study on Abstract Interpretation and 'Validating Microcode Algebraically', *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), (Ellis Horwood, 1987), 199–218.

[Naur, 1965] P. Naur: Checking of Operand Types in ALGOL Compilers, *BIT* **5** 151–163 (1965).

[Nielson, 1982] F.Nielson: A Denotational Framework for Data Flow Analysis, *Acta Informatica* **18** (1982) 265–287.

[Nielson, 1983] F.Nielson: Towards Viewing Nondeterminism as Abstract Interpretation, *Foundations of Software Technology & Theoretical Computer Science* **3** (1983).

[Nielson, 1984] F.Nielson: Abstract Interpretation using Domain Theory, *Ph.D.-thesis CST-31-84*, (University of Edinburgh, Scotland, 1984).

[Nielson, 1985a] F.Nielson: Program Transformations in a Denotational Setting, *ACM Transactions on Programming Languages and Systems* **7** (1985) 359–379.

[Nielson, 1985b] F.Nielson: Tensor Products Generalize the Relational Data Flow Analysis Method, *Proc. 4th Hungarian Computer Science Conference* (1985) 211–225.

[Nielson, 1986a] F.Nielson: Abstract Interpretation of Denotational Definitions, *Proc. STACS 1986*, Lecture Notes in Computer Science **210** (Springer-Verlag, 1986) 1–20.

[Nielson, 1986b] F.Nielson: Expected Forms of Data Flow Analysis, *Programs as Data Objects*, Lecture Notes in Computer Science **217** (Springer-Verlag, 1986) 172–191.

[Nielson, 1986c] F.Nielson: A Bibliography on Abstract Interpretation, *ACM Sigplan Notices* **21 (5)** (1986) 31–38 and *Bulletin of the EATCS* **28** (1986) 45–52.

[Nielson, 1987] F.Nielson: Towards a Denotational Theory of Abstract Interpretation, *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), (Ellis Horwood, 1987), 219–245.

[Nielson, 1988a] F.Nielson: Strictness Analysis and Denotational Abstract Interpretation, *Information and Computation* **76** (1988) 29–92. Also see *Proc.14th ACM Symposium on Principles of Programming Languages* (1987) 120–131.

[Nielson, 1988b] H.R.Nielson, F.Nielson: Automatic Binding Time Analysis for a Typed λ-calculus, *Science of Computer Programming* **10** (1988) 139–176. Also see *Proc. 15th ACM Symposium on Principles of Programming Languages* (1988) 98–106.

[Nielson, 1988c] F.Nielson, H.R.Nielson: 2-level λ-lifting, *Proc. ESOP 1988*, Lecture Notes in Computer Science **300** (Springer-Verlag, 1988) 328–343.

[Nielson, 1988d] F.Nielson, H.R.Nielson: The TML-approach to compiler-compilers, *report ID-TR 1988-47*, (The Technical University of Denmark, 1988).

[Nielson, 1989] F.Nielson: Two-Level Semantics and Abstract Interpretation, *Theoretical Computer Science — Fundamental Studies* **69** (1989) 117–242.

[Nielson, 1992a] H.R.Nielson, F.Nielson: *Semantics with Applications: A Formal Introduction for Computer Science*, Wiley, 1992.

[Nielson, 1992b] F.Nielson, H.R.Nielson: *Two-Level Functional Languages*, Cambridge Tracts in Theoretical Computer Science **34**, Cambridge University Press, 1992.

[Nilsson, 1991] U. Nilsson: Abstract Interpretation: a Kind of Magic, in *Programming Language Implementation and Logic Programming*, edited by J. Maluszynski, M.Wirsing, Lecture Notes in Computer Science **528**, 299-309 (Springer-Verlag, 1991).

[Paulson, 1984] L.Paulson: Compiler Generation from Denotational Semantics, *Methods and Tools for Compiler Construction*, B.Lorho (ed.), (Cambridge University Press, 1984) 219–250.

[Plaisted, 1984] D. Plaisted: The Occur-Check Problem in Prolog, in Proceedings of the International Symposium on Logic Programming, New Jersey (1984).

[Plotkin, 1976] G.D.Plotkin: A powerdomain construction, *SIAM Journal on Computing* **5** (1976) 452–487.

[Plotkin, 1980] G.D.Plotkin: Lambda Definability in the Full Type Hierarchy, *To H.B.Curry: Essays on Combinatorial Logic, Lambda Calculus and Formalism*, J.P.Seldin and J.R.Hindley (eds.), (Academic Press, 1980).

[Plotkin, 1981] G.D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981. (Reprinted 1991.)

[Pitt, 1986] *Category Theory and Computer Programming*, edited by D.Pitt, S.Abramsky, A.Poigne and D.Rydeheard, Lecture Notes in Computer Science **240** (Springer-Verlag, 1986).

[Reynolds, 1969] J. Reynolds: Automatic Computation of Data Set Definitions, *Information Processing* **68**, 456-461 (North-Holland, 1969).

[Reynolds, 1974] J.C.Reynolds: On the Relation between Direct and Continuation Semantics, *Proc. ICALP 1974*, Lecture Notes in Computer Science **14** (Springer-Verlag, 1974) 141–156.

[Romanenko, 1990] S.A. Romanenko: Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, 341–360. Springer-Verlag, 1990.

[Rosendahl, 1989] Mads Rosendahl: Automatic Complexity Analysis. In Functional Programming Languages and Computer Architecture, in *FPCA '89 Conference Proceedings,* 144–156, ACM Press, 1989.

[Schmidt, 1986] D.A.Schmidt: *Denotational Semantics: A Methodology for Language Development*, (Allyn and Bacon, 1986).

[Sestoft, 1989] P. Sestoft: Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.

[Sestoft and Argo, 1989] Peter Sestoft and Guy Argo, Detecting Unshared Expressions in the Improved Three Instruction Machine, November 1989. DIKU, University of Copenhagen, Denmark. Submitted for publication.

[Sharir, 1981] M. Sharir, A. Pnueli: Two Approaches to Interprocedural Data Flow Analysis, in *Program Flow Analysis*, eds. S.S. Muchnick, N.D. Jones (Prentice-Hall 1981).

[Shivers, 1991] O. Shivers: The Semantics of Scheme Control-flow Analysis, in *Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices **26**, 9 (ACM 1991).

[Sintzoff, 1972] M. Sintzoff, Calculating Properties of Programs by Valuations on Specific Models, *Proceedings ACM Conference on Proving Assertions abourt Programs*, 203-207 (1972).

[Søndergaard, 1986] H. Søndergaard: An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction, *Proceedings of the European Symposium on Programming* (Springer-Verlag, 1986).

[Steffen, 1987] B.Steffen: Optimal Run Time Optimization — Proved by a New Look at Abstract Interpretations, TAPSOFT, *Lecture Notes in Computer Science* **249** (Springer-Verlag, 1987) 52-68.

[Stoy, 1977] J.E. Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[Stransky, 1990] J. Stransky: A Lattice for Abstract Interpretation of Dynamic (LISP-like) Structures, *Information and Computation* (1990). Research report LIX/RR/90/03.

[Wadler, 1987] P. Wadler, R.J.M. Hughes: Projections for Strictness Analysis. In Gilles Kahn, editor, *FPCA'87. Functional Programming and Computer Architecture, Portland, Oregon, Sept., 1987. (Lecture Notes in Computer Science, vol. 274)*, 385–4078. Springer-Verlag, 1987.

[Wadler, 1988] P. Wadler: Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP'88. 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300)*, 344–358. Springer-Verlag, 1988.

[Wegbreit, 1975] B. Wegbreit: Property Extraction in Well-founded Property Sets, *IEEE Transactions on Software Engineering* **SE-1**, 270-285 (1975).