1

An introduction to abstract interpretation

Samson Abramsky and Chris Hankin, Imperial College, London, UK

1. INTRODUCTION

A significant proportion of the code in most modern production compilers is devoted to the optimization of generated code. All too often the run-time behaviour of the optimized program is inconsistent with the pre-optimization behaviour; in other words, the optimization has affected the semantics of the program as well as the pragmatics. This problem normally arises because insufficient rigour has been applied to the correctness proof of the optimization. For programming languages with defined mathematical semantics there is a growing set of tools that provides a basis for semantically correct transformation; one such tool is abstract interpretation. This book is mainly, but not exclusively, devoted to the presentation of various uses of abstract interpretation in the compile-time analysis of declarative programming languages. Declarative languages are being increasingly used as the basis for rapid prototyping systems; tools such as abstract interpretation increase the possibility of turning the *runnable specifications* into efficient, production programs.

In this chapter, we will review some of the major stages in the development of the field. We will also present some of the basic ideas from denotational semantics that are required by later chapters. However, first we introduce two non-computing examples of abstract interpretation which will serve to establish the principles underlying the approach.

Suppose that we are faced with the prospect of travelling somewhere; one decision that must be made is whether we walk, drive or fly. Rather than make our decision by trial and error, we use a property of the journey, the distance (which can be measured on a map), to decide which is the most appropriate mode of transport. The map is an abstract representation of the journey and by measuring the distance we abstract the travelling process.

A second, more formal, example is the use of the *rule of signs* to determine the sign of the result of a calculation. Asked for the sign of

$$151515*(-235)$$

we immediately answer that the result is negative. Instead of performing the multiplication, we use the rule of signs that tells us that multiplying a positive number by a negative number always gives us a negative result. This second example is a bit closer to programming than the first and so we shall study it in a bit more detail. To produce our answer, we had to translate the task into the following form:

$$+ \mathbin{\underset{-}{*}} -$$

where $\underset{-}{*}$ is the rule of signs version of multiplication:

$$\begin{gathered}0 \mathbin{\underset{-}{*}} + = 0 \mathbin{\underset{-}{*}} - = + \mathbin{\underset{-}{*}} 0 = - \mathbin{\underset{-}{*}} 0 = 0\\ + \mathbin{\underset{-}{*}} + = - \mathbin{\underset{-}{*}} - = +\\ + \mathbin{\underset{-}{*}} - = - \mathbin{\underset{-}{*}} + = -\end{gathered}$$

and then execute this, much simpler, calculation.

So far we have not considered the correctness of the interpretations but it should be clear that we *can* get completely accurate answers in both examples. The situation becomes much less clear if we extend the calculations of the second example to include addition. The first few rules of the rule of signs version of addition present little problem:

$$\begin{gathered}+ \pm 0 = 0 \pm + = +\\ - \pm 0 = 0 \pm - = -\\ + \pm + = +\\ - \pm - = -\end{gathered}$$

but the rest are slightly problematical:

$$\begin{gathered}+ \pm - = ??\\ - \pm + = ??\end{gathered}$$

If we arbitrarily choose a sign (0, + or −) in place of ?? we shall get incorrect answers some of the time because the answer to the real calculation will depend on the actual magnitude of the two numbers. How can we characterize a correct choice for ??? In order to do this we must consider what the signs in the abstract calculation represent:

$$\begin{aligned}0 &= \{0\}\\ + &= \{n \mid n > 0\}\\ - &= \{n \mid n < 0\}\end{aligned}$$

then the abstract calculation is correct if the real answer is a member of the set that the abstract answer represents. If this is the case, we say that the abstract interpretation if *safe*. By using ?? to represent the set of integers, we get a safe version of addition by adding the rules:

$$?? \pm s = s \pm ?? = ?? \pm ?? = ?? \qquad \text{where } s \in \{0, +, -\}$$

We leave it as an exercise for the reader to extend the rule of signs for multiplication.

How is abstract interpretation useful in computing? Many of the traditional optimizations based on control-flow and data-flow analysis fit within the abstract interpretation framework. Some of the particular analyses that are important for declarative languages are listed below.

Strictness Analysis: An analysis that allows the optimization of lazy functional programs by identifying the parameters that can be passed by value thus avoiding the need to build closures and opening up opportunities for parallel evaluation. Chapters 4 and 12 address this area and we consider two other approaches later in this chapter.

In-place Update Analysis: This analysis allows us to determine the points in a program at which it is safe to destroy a data object because there are no longer any references to it. Results in this area are reported by Hudak. A notable result is that, for the first time, a functional version of the quicksort algorithm can be made to run in linear space.

Relevant Clause Analysis: In many of the prototype fifth generation architectures programs are able to make non-local access to function definitions. This implies that there is a communication overhead associated with program execution. By using this particular analysis it is possible to identify the parts of the function definition that are relevant to a particular application and thus reduce the overhead.

Mode Analysis: Significant performance improvements can be achieved in PROLOG interpreters if it is known how the logical variables are used in a relation (i.e. as input or output variables or a mixture of the two). This problem is addressed by both Mellish's chapter and also Jones and Sondergaard's chapter.

As the declarative language community become more aware of abstract interpretation, new applications are being discovered. A common feature of the compiler optimization applications is that they abstract properties of programs that are essentially undecidable. Thus the issue of correctness becomes critical. Optimization based on safe abstract interpretations are probably correct. Translated into our examples safety implies that:

- If strictness analysis determines that a function is strict in an argument then it definitely is but the analysis will fail to detect some parameters that could be passed by value
- If in-place update analysis indicates that we can destructively update a cell then we can but we will still copy some objects that could have been destroyed
- Relevant clause analysis will cause us to communicate a superset of the code that is actually needed for a particular application
- Mode analysis will sometimes fail to detect that a logical variable is used exclusively as an input(output) variable

But even with this unavoidable inaccuracy, there are still significant performance payoffs.

In the rest of this chapter we shall try to convey in a much more precise way how abstract interpretation is used in computing and in particular in declarative programming. We start with some elementary domain theory, we then present overviews of a number of important contributions to the field. The Cousots' work is included since it is seminal; although their framework was developed for the analysis of flowchart programs it served to establish the approach. Mycroft developed the first applications of abstract interpretation to functional languages and our own work is a natural extension of his. Throughout the chapter we have provided appropriate cross references to later chapters.

2. MATHEMATICAL PRELIMINARIES

A number of chapters assume some familiarity with domain theory and denotational semantics. In order to make the book reasonably self-contained, we shall briefly review some basic concepts in domain theory. For excellent textbook presentations of denotational semantics, see [Gor79b], [Sch86] and [Sto77].

Firstly, what problems was domain theory introduced to solve? The essential one was to give meaning to recursive definitions of

(1) programs
(2) data types

A naive approach to denotational semantics would attempt to base itself on sets and functions. To see how this leads to problems, consider the definition:

$$f\colon Bool \rightarrow Bool$$
$$f(x) = not\ f(x) \tag{1}$$

which (with minor syntactic modifications) is a valid program in any language allowing recursive functions or procedures. If we regard the type *Bool* as a set:

$$Bool = \{tt, ff\}$$

and f as a function on this set, then it is easy to see that no such function satisfies the equation (1), and so we have not succeeded in defining anything at all! Thinking computationally about this example, we see that what in fact is going on is that f is a non-terminating program. This suggests the device of introducing partial elements to model non-termination, and hence restoring f to the status of a well-defined function:

$$\begin{tikzcd} Bool_\perp \equiv tt \arrow[dr, dash] & & ff \arrow[dl, dash] \\ & \perp & \text{non-termination} \end{tikzcd}$$

if we extend *not* in the obvious way by

$$not\ \perp = \perp$$

then we see that, for $x \in Bool_\perp$

$$f(x) = \perp = not\ \perp = not\ f(x)$$

and we have recovered a consistent definition.

The basic strategy of domain theory is thus to expand the class of definitions

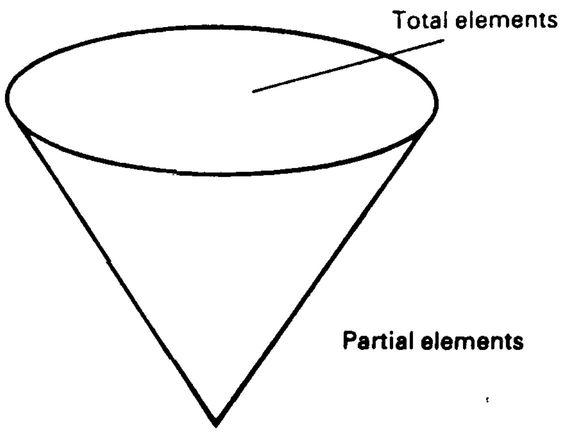

Fig. 1.

(in particular, recursive definitions) to which we can give meaning by replacing inconsistency with non-termination, which is *objectified* or *denotationalized* into partial elements.

Thus we may think of domains as sets plus an infrastructure of partial elements representing partially defined approximations to the ordinary *total* elements. This imposes a structure of partial elements approximating more defined ones, which is usually formulated as a partial ordering (the information or approximation ordering). Computable functions between domains preserve this structure.

On this basis, we can formulate some axioms for domains:

(Axiom 1) Domains are partially ordered sets $(D, \sqsubseteq)$ with least elements $\perp_D$ with

$$\perp_D \sqsubseteq d,\ \forall d \in D$$

(Axiom 2) Computable functions between domains are monotonic (preserve the ordering)

$$f{:}D \rightarrow E \text{ is monotonic} \equiv \forall d, e \in D.\, d \sqsubseteq e \Rightarrow fd \sqsubseteq fe$$

Moreover, we want to compute meanings in domains as 'limits of finite approximations'. This will require that

(1) suitable limits exist
(2) computable functions preserve them

which gives the following two axioms:

(Axiom 3) Domains are complete partial orders (cpos):
for each increasing sequence (or chain) $\{x_n\}$ in D (i.e. $x_1 \sqsubseteq \cdots x_n \sqsubseteq x_{n+1} \sqsubseteq \cdots$) the least upper bound of this sequence, written

$$\bigsqcup_{n=0}^{\infty} x_n \in D$$

exists

(The defining properties of the least upper bound are

(1) $\forall n. x_n \sqsubseteq \bigsqcup_{n=0}^{\infty} x_n$

(2) $\forall d \in D. (\forall n. x_n \sqsubseteq d) \Rightarrow \bigsqcup_{n=0}^{\infty} x_n \sqsubseteq d$)

(Axiom 4) Computable functions are continuous (preserve limits).

$f: D \to E$ is continuous $\equiv$ for every chain $\{x_n\}$ in D, $f\left(\bigsqcup_{n=0}^{\infty} x_n\right) = \bigsqcup_{n=0}^{\infty} f(x_n)$

2.1 Examples of domains

(1) Given a set S, let $S_\perp = S \cup \{\perp\}$ and define a ordering $\sqsubseteq$ on $S_\perp$ by:

$$x \sqsubseteq y \equiv x = \perp \text{ or } x = y$$

The least element is $\perp$. Any chain has the form:

$$\underbrace{\perp \sqsubseteq \ldots \sqsubseteq \perp}_{n \geqslant 0} \sqsubseteq s \sqsubseteq \ldots \sqsubseteq s \sqsubseteq \ldots$$

or

$$\perp \sqsubseteq \perp \sqsubseteq \perp \sqsubseteq \ldots$$

This construction of flat domains is a recipe for making domains out of sets by adding a single partial element to represent non-termination.

(2) A domain of tapes or output streams (see Fig. 2). This domain comprises all finite and infinite sequences of 0s and 1s, ordered by:

$$s \sqsubseteq t \equiv s \text{ is an initial subsequence of } t$$

Examples of this ordering are:

$0 \sqsubseteq 01 \sqsubseteq 010$

$\bigsqcup_{n=0}^{\infty} 0^n = 0^\omega$ (infinite sequences of zeroes)

(3) If D and E are cpos, then

$$[D \to E]$$

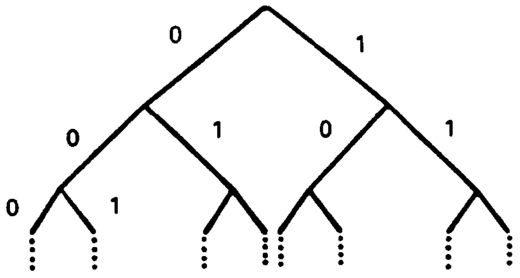

Fig. 2.

the set of all continuous functions from D to E ordered *pointwise* by:

$$f \sqsubseteq g \equiv \forall d \in D.\, f(d) \sqsubseteq g(d)$$

is a cpo. The least element is

$$\lambda x \in D.\, \bot_E$$

Given a chain of functions:

$$\{f_n\}$$

in $[D \rightarrow E]$, the least upper bound is defined pointwise by:

$$\left(\bigsqcup_{n=0}^{\infty} f_n\right)(d) = \bigsqcup_{n=0}^{\infty} f_n(d)$$

(4) If D and E are cpos, then

$$D \times E = \{\langle d,e\rangle \mid d \in D \textbf{ and } e \in E\}$$

the Cartesian product, ordered by:

$$\langle d,e\rangle \sqsubseteq \langle d',e'\rangle \equiv d \sqsubseteq d' \textbf{ and } e \sqsubseteq e'$$

is a cpo. The least element is

$$\langle \bot_D, \bot_E \rangle$$

Given a chain $\{\langle d_n, e_n\rangle\}$ in $D \times E$, $\{d_n\}$ forms a chain in D and $\{e_n\}$ forms a chain in E and:

$$\bigsqcup_{n=0}^{\infty} \langle d_n, e_n \rangle = \left\langle \bigsqcup_{n=0}^{\infty} d_n, \bigsqcup_{n=0}^{\infty} e_n \right\rangle$$

2.2 Fixed points

The axioms presented earlier may seem very abstract. However as an immediate and spectacular payoff, we get:

The Fixed Point Theorem If $f: D \rightarrow D$ is a continuous function on a domain D, it has a least fixed point $d \in D$, satisfying:

(1) $f(d) = d$
(2) $\forall e \in E.\, f(e) = e \Rightarrow d \sqsubseteq e$

Moreover, d is defined by:

$$d = \bigsqcup_{n=0}^{\infty} f^n(\bot_D)$$

(where $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$)

Proof Firstly,

$$\bot_D \sqsubseteq f(\bot_D) \qquad (\bot_D \text{ is the least element})$$

Moreover,

$$f^n(\bot_D) \sqsubseteq f^{n+1}(\bot_D) \Rightarrow f^{n+1}(\bot_D) \sqsubseteq f^{n+2}(\bot_D) \qquad (f \text{ is monotonic})$$

and so by induction the sequence $\{f^n(\bot_D)\}$ is a chain.

Now

$$f\left(\bigsqcup_{n=0}^{\infty} f^n(\bot_D)\right) = \bigsqcup_{n=0}^{\infty} f^{n+1}(\bot_D) \qquad (f \text{ is continuous})$$

$$= \bigsqcup_{n=0}^{\infty} f^n(\bot_D)$$

and so $\bigsqcup_{n=0}^{\infty} f^n(\bot_D)$ is a fixed point of f.

Finally, suppose e is a fixed point of f, that is $f(e) = e$. We show that $f^n(\bot_D) \sqsubseteq e$ of all n by induction. The basis is just:

$$f^0(\bot_D) = \bot_D \sqsubseteq e$$

For the induction step,

$$f^n(\bot_D) \sqsubseteq e \Rightarrow f(f^n(\bot_D)) \sqsubseteq f(e) = e$$

This shows that e is an upper bound of the chain $\{f^n(\bot_D)\}$ and hence,

$$\bigsqcup_{n=0}^{\infty} f^n(\bot_D) \sqsubseteq e \qquad \blacksquare$$

Note that this proof uses all of our four axioms for domains.

The fixed point theorem enables us to interpret recursive definitions

$$x = f(x)$$

as least fixed points and thus solves the problem with which we started. It also forms the basis for a number of practical algorithms in abstract interpretation, as we shall see.

3. THE COUSOTS – SOME BASIC NOTATION

The first people to attempt to construct a rigorous framework for abstract interpretation were Patrick and Radhia Cousot. Many of the basic concepts and definitions that recur in later work may be traced, albeit in modified form, to the Cousots' work. They were primarily interested in the flow analysis of flowchart programs although their work provided the inspiration for Mycroft's thesis work and much that has followed since.

A program is represented by a graph; the nodes correspond to program operations and the arcs record control flow. To fix notation we follow the Cousots and use a simple language in which each node is labelled by one of the following classes of operation: Entry, Assignment, Test, Junction and Exit. Each node has a set of successor nodes and a set of predecessor nodes associated with it; these sets are given by the following functions:

$$n\text{-}succ,\ n\text{-}pred : Nodes \to 2^{Nodes}$$

that satisfy:

$$m \in n\text{-}succ(n) \Leftrightarrow n \in n\text{-}pred(m)$$

For each type of node there are constraints on the cardinalities of the successor and predecessor sets; for example, for Test nodes we have:

$$|n\text{-}pred(n)| = 1$$
$$|n\text{-}succ(n)| = 2 \qquad \textit{a true and a false successor}$$

For Test nodes we will also have the functions *n-succ-t* and *n-succ-f* which select the true and false successors respectively. The set of *Arcs* of a program is a subset of *Nodes* × *Nodes* defined by:

$$Arcs = \{\langle m, n\rangle \mid (m \in Nodes) \text{ and } (n \in n\text{-}succ(m))\}$$

The functions *a-succ*, *a-pred*, *a-succ-t* and *a-succ-f* are defined analogously to the above. A program is then represented as the directed graph ⟨*Nodes*, *Arcs*⟩; for example the ubiquitous factorial program (see Fig. 3).

The semantics of such programs can be given in the following way. Firstly we need a state; this is represented by an arc (the program counter) and a memory (called the environment in the Cousots' work) that records the

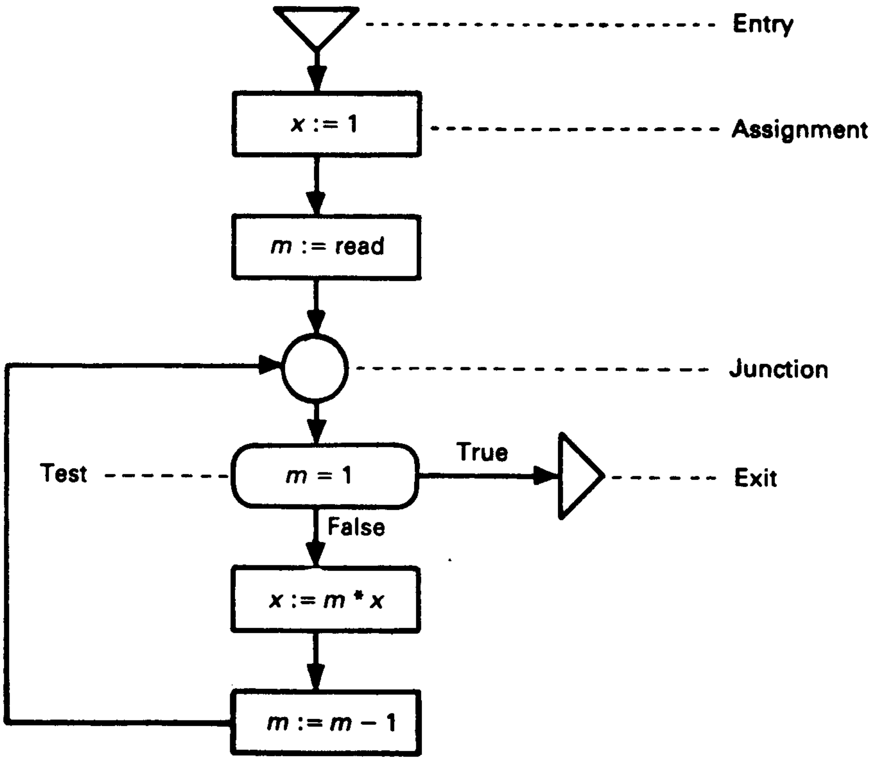

Fig. 3.

current bindings of identifiers to values. Thus we define

$$\begin{aligned}&Values = Bool + Num + \cdots\\&Env = Ident \rightarrow Values\\&States = Arcs_{\perp} \times Env\end{aligned}$$

where *Ident* is a set of identifiers and we use the standard domains of booleans and numbers and + is disjoint sum. (The Cousots require that the various sets involved be complete lattices but this is an unnecessary restriction.) The semantics are given by a state transition function:

$$next: States \rightarrow States$$

$$\begin{aligned}next(\langle m,n\rangle,\rho) = \mathbf{case}\ n\ \mathbf{in}\ \\ &Assignments{:}(a\text{-}succ(n),\rho[val[\![expr(n)]\!]\rho/id(n)])\\ &Tests{:}cond(val[\![test(n)]\!]\rho,(a\text{-}succ\text{-}t(n),\rho),(a\text{-}succ\text{-}f(n),\rho))\\ &Junctions{:}(a\text{-}succ(n),\rho)\\ &Exits{:}(\langle m,n\rangle,\rho)\\ \mathbf{esac}\end{aligned}$$

where

(1) *expr*, *id* and *test* are the obvious syntactic selector functions
(2) *val* is a semantic function

$$val: Expressions \rightarrow Env \rightarrow Values$$

that gives meanings to expressions
(3) $\rho[v/I]$ where $\rho \in Env, I \in Ident$ and $v \in Values$ is the result of updating ρ at I with v

$$\begin{aligned}\rho[v/I]I' &\rightarrow v, I = I'\\ &\rho I', \text{otherwise}\end{aligned}$$

For simplicity we shall consider programs which have a single input parameter which is associated with the identifier *read*. The initial states are characterized by:

$$Istates = \{(a\text{-}succ(m),\lambda i \in Ident.\mathbf{if}\ i = read\ \mathbf{then}\ initial\text{-}value\ \mathbf{else}\ \perp) \mid m \in Entries,\ initial\text{-}value \in Values\}$$

and the meaning of the program is the solution of the equation:

$$P = next \circ P$$

which is given by

$$fix(\lambda f.next \circ f)$$

In the Cousots' framework all abstract interpretations are defined in terms of the *static semantics* (the *collecting* interpretation of Mycroft). The static semantics is in some sense the *least* abstract of the abstract interpretations; it collects together all of the environments that might be associated with a program point (arc) during program execution. Any property of interest can be inferred by analysis of these sets.

In the static semantics a context, which is a member of the powerset of environments, is associated with each arc:

$$\mathit{Contexts} = 2^{\mathit{Env}}$$

The context associated with a particular arc, q, is defined as:

$$\{e \mid \exists n \geqslant 0,\ \exists i \in \mathit{Istates},\ \langle q,e\rangle = \mathit{next}^n(i)\}$$

Contexts is a complete lattice with $\varnothing$ as the bottom element, *Env* as the top, $\subseteq$ as the ordering and $\cup$ and $\cap$ as join and meet operations.

The static semantics is uncomputable because it gives exact information about program properties. We therefore consider *abstract* interpretations which *are* computable at the price of giving less precise information. An abstract interpretation is given by a complete lattice of abstract contexts and an interpretation function. But which abstract interpretations are permissible? The Cousots establish a correspondence between concrete and abstract contexts via a pair of functions α (abstraction) and γ(concretization) that have the following properties:

$$\alpha : \mathit{Contexts} \rightarrow \mathit{Abstract}$$
$$\gamma : \mathit{Abstract} \rightarrow \mathit{Contexts}$$
$$\forall x \in \mathit{Abstract}.\ x = \alpha(\gamma(x))$$
$$\text{or } \alpha \circ \gamma = \mathit{identity}_{\mathit{Abstract}}$$
$$\forall x \in \mathit{Contexts}.\ x \sqsubseteq \gamma(\alpha(x))$$
$$\text{or } \gamma \circ \alpha \sqsupseteq \mathit{identity}_{\mathit{Contexts}}$$

This defines (α, γ) as an *adjoined pair* of functions. If these functions can be established for a particular abstract interpretation, then the *correctness* of the interpretation follows. (Correctness means that the concretized abstract context includes the concrete context – recall the rule of signs example).

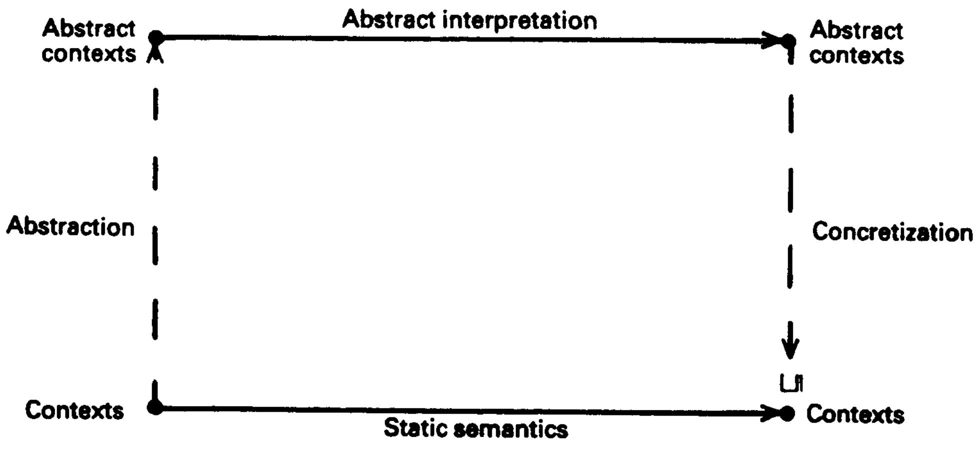

Fig. 4.

As a first example we extend the rule of signs to flowchart programs. It simplifies the treatment to consider programs which only use one identifier, the input parameter *read*, but the reader should have little difficulty in generalizing the example. An immediate implication of restricting ourselves to one identifier is that we can treat the concrete contexts as sets of integers rather than environments. The abstract contexts are signs drawn from the following lattice:

$$\begin{tikzcd}
& ?? \arrow[dl, dash] \arrow[dr, dash] & \\
- \arrow[dr, dash] & & + \arrow[dl, dash] \\
& 0 \arrow[d, dash] & \\
& \bot &
\end{tikzcd}$$

The interpretation is then given by

```
$ros: Arcs_\bot \times (Arcs_\bot \rightarrow Sign) \rightarrow Sign$
$ros(\langle m,n \rangle, S) =$ case $m$ in
        $Entries$: $sign\ of\ the\ initial\ value$
        $Junctions$: $\mathbf{join}_{q \in a\text{-}pred(m)} S(q)$
        $Tests$: case $\langle m,n \rangle$ in
            $\{a\text{-}succ\text{-}t(m)\}$: $\bigsqcup \{s \mid s \sqsubseteq S(a\text{-}pred(m))$
                                    and $tval[\![test(m)]\!] s \sqsupseteq true\}$
            $\{a\text{-}succ\text{-}f(m)\}$: $\bigsqcup \{s \mid s \sqsubseteq S(a\text{-}pred(m))$
                                    and $tval[\![test(m)]\!] s \sqsupseteq false\}$
            esac
        $Assignments$: $sign[\![expr(m)]\!]\ S(a\text{-}pred(m))$
            esac
```

where **join** is the usual join operation on lattices and *tval* and *sign* are non-standard semantic functions

$$sign: Expressions \rightarrow Sign \rightarrow Sign$$
$$tval: Expressions \rightarrow Sign \rightarrow Truval$$

that apply the rule of signs described in the Introduction to evaluate expressions. (In a more general solution the *Sign* parameter for each function would be replaced by a sign environment, *Ident* → *Sign*). The domain of truth values, *Truval*, is a complete lattice:

$$\begin{tikzcd}
& truefalse \arrow[dl, dash] \arrow[dr, dash] & \\
true \arrow[dr, dash] & & false \arrow[dl, dash] \\
& \bot &
\end{tikzcd}$$

The top element is the value of expressions that do not have a clearcut value, for example:

$$+ \geqslant +$$

The abstraction map is defined in the following way:

$$\begin{aligned}
&\alpha{:}\,Contexts \to Sign\\
&\alpha(C) = \bot, C = \{\,\}\\
&\qquad\quad 0, C = \{0\}\\
&\qquad\quad +, C = \{c \mid c \geqslant 0\}\\
&\qquad\quad -, C = \{c \mid c \leqslant 0\}\\
&\qquad\quad ??, \text{ otherwise}
\end{aligned}$$

and the concretization map is given by

$$\begin{aligned}
&\gamma{:}\,Sign \to Contexts\\
&\gamma(S) = \{\,\}, S = \bot\\
&\qquad\quad \{0\}, S = 0\\
&\qquad\quad \{s \mid s \geqslant 0\}, S = +\\
&\qquad\quad \{s \mid s \leqslant 0\}, S = -\\
&\qquad\quad \{s \mid s \text{ an integer}\}, \text{ otherwise}
\end{aligned}$$

It is a trivial exercise to show that these satisfy the adjoinedness property.

How do we use such an abstract interpretation? We want to annotate every arc in the program with an element of Sign that safely represents the possible values that the identifier might have at that point in the program. Initially we have no information; all arcs have $\bot$ associated with them. After applying the interpretation once we shall know the sign of the initial value. Repeated application of the interpretation will propagate sign information throughout the program. We have just described the usual process of iterating towards a fixed point. The required information can be found as the fixed point of the following functional:

$$\begin{aligned}
&F{:}(Arcs_\bot \to Sign) \to Arcs_\bot \to Sign\\
&F = \lambda S.\lambda r.ros(r,S)
\end{aligned}$$

using:

$$\lambda r \in Arcs_\bot.\bot$$

to start the iteration.

A second example which uses a simpler abstract domain is an analysis which determines which nodes in a program are *reachable*. We include this analysis because it is closely related to the strictness analysis problem that is discussed later. The abstract contexts are elements of the two-point domain:

$$\begin{tikzcd}
\top \arrow[d, no head] \\
\bot
\end{tikzcd}$$

and the interpretation is given by:

$$\begin{aligned}
&\boldsymbol{reach}{:}\,\boldsymbol{Arcs_\bot \times (Arcs_\bot \to Reach) \to Reach}\\
&\quad \boldsymbol{reach(\langle m,n\rangle, R) = \textbf{case } m \textbf{ in}}
\end{aligned}$$

```
        Entries: $\top$
        Junctions: join$_{q \in a\text{-}pred(m)} R(q)$
        Tests: case $\langle m,n \rangle$ in
                $\{a\text{-}succ\text{-}t(m)\}: \bigsqcup \{r \mid r \sqsubseteq R(a\text{-}pred(m))$
                                        and $tval[\![test(m)]\!]r \sqsupseteq true\}$
                $\{a\text{-}succ\text{-}f(m)\}: \bigsqcup \{r \mid r \sqsubseteq R(a\text{-}pred(m))$
                                        and $tval[\![test(m)]\!]r \sqsupseteq false\}$
                esac

        Assignments: $R(a\text{-}pred(m))$
    esac
```

where *tval* is the obvious modification of the function of the same name used in the first example. The abstraction and concretization maps are defined as follows:

$$
\begin{aligned}
&\alpha: Contexts \rightarrow Reach\\
&\alpha(C) = \bot, C = \{\,\}\\
&\qquad \top\text{, otherwise}\\
&\gamma: Reach \rightarrow Contexts\\
&\gamma(R) = \{\,\}, R = \bot\\
&\qquad \{r \mid r \text{ an integer}\}\text{, otherwise}
\end{aligned}
$$

Again it is a trivial exercise to show that this pair of maps satisfy the adjoinedness property. The reader is invited to work through some examples of these two interpretations.

A notable omission from the foregoing discussion is the distinction between *forwards* and *backwards* analysis. In the examples we showed analyses that propagate information forwards from the entry points; we could equally well start with exit properties and use these to infer input constraints. We shall not dwell on this issue here; the reader is referred to Chapter 4 and Burn's thesis where backwards analysis is used to perform strictness analysis on functional programs that operate on structured data.

Nielson's work may be seen as a generalization of the Cousots'; he provides a framework that applies to any language that can be given a denotational semantics (rather than just flowcharts). As stated at the beginning of this section, the notation introduced by the Cousots pervades the subject but Chapters 2, 5 and 9 provide alternative frameworks. Chapters 6 and 8, on analysing PROLOG programs, are closely based on the Cousots' work.

4. MYCROFT – ABSTRACT INTERPRETATION IN THE APPLICATIVE IDIOM I

At first sight it is not clear how to apply the ideas of the last section to functional programs. There is no clear notion of program point in functional programs and the powerset framework does not allow us to perform certain analyses that are of interest. Graph reduction implementations of functional languages are self-optimizing to a certain extent so that, rather than the classical analyses, questions of termination become much more interesting (cf. strictness analysis). Based on these observations, Mycroft gives cogent reasons for moving to the use of *powerdomains* for the static semantics rather than powersets. Indeed, Mycroft shows that the powerset interpretation is an abstraction of the *collecting* interpretation (Mycroft's term for static semantics).

4.1 Powerdomains

Powerdomains inherit structure from the underlying domains. Distinct sets from the powerset become identified in the powerdomain; which sets are identified depends on the particular powerdomain construction that we use. A domain element, x, approximates another, y, ($x \sqsubseteq y$) if every *property* satisfied by x is also satisfied by y. Properties can be represented as the set of objects that satisfy them. More formally, if we have a *characteristic* function:

$$f : D \to 2 \qquad (D \text{ any domain, } 2 = \{\bot, \top\})$$

then we can define property P by

$$P = \{d \in D \mid f\,d = \top\}$$

We shall be interested in *finitely observable* properties; these are properties which have continuous characteristic functions (these sets are the Scott Open sets). For domain element x, we write *x satisfies property P* if x is a member of the above set.

There are two ways of extending this notion of satisfaction of a property, P, to sets of elements, S:

(1) $\forall s \in S.\ s \in P$

i.e. $S \subseteq P$, read S ***must satisfy*** P

(2) $\exists s \in S.\ s \in P$

i.e. $S \cap P \neq \varnothing$, read S ***may satisfy*** P

From this we can derive three orderings:

(1) $R \sqsubseteq_1 S \equiv \forall$ observable P. R may satisfy $P \Rightarrow S$ may satisfy P
(2) $R \sqsubseteq_2 S \equiv \forall$ observable P. R must satisfy $P \Rightarrow S$ must satisfy P
(3) $R \sqsubseteq_3 S \equiv R \sqsubseteq_1 S$ and $R \sqsubseteq_2 S$

and each has an equivalence associated with it:

$$R =_i S \equiv R \sqsubseteq_i S \text{ and } S \sqsubseteq_i R \qquad i = 1, 2 \text{ or } 3$$

For a powerdomain construction we may choose one of these orderings with elements that are canonical representatives for the associated equivalence classes. In particular, there is a largest set in each equivalence class (with respect to set inclusion). In the powerdomain based on $\sqsubseteq_1$, these sets are exactly the Scott Closed subsets which we therefore choose as the canonical representatives. A Scott Closed set is a set X such that:

(1) $\forall$chains $\{y_n\} \sqsubseteq X, \bigsqcup \{y_n\} \in X$
(2) $\forall y \in D$ such that $y \sqsubseteq x$ for some $x \in X$, $y \in X$. (X is left-closed)

Using these elements $\sqsubseteq_1$ becomes $\subseteq$. This is the Hoare (or lower) powerdomain. Similar descriptions can be given for the other two standard powerdomain constructions; the Smyth (or upper) powerdomain and the Plotkin powerdomain.

We now return to Mycroft's thesis work.

4.2 Mycroft's framework

Mycroft uses a language of first-order recursion equations. An abstract syntax for such a language is:

$$Prog ::= Exp \textbf{ where } (Fun(Var_1 \ldots Var_k) = Exp)^+$$
$$Exp ::= Atom \mid Var \mid Basic(Exp_1, \ldots, Exp_m) \mid Fun(Exp_1, \ldots, Exp_k)$$

A schematic semantics can be given in the following way:

$$\mathbf{M}: Prog \to D$$
$$\mathbf{M}[\![E \textbf{ where } F_1(x_1, \ldots, x_k) = E_1 \ldots]\!] = \mathbf{E}[\![E]\!]\rho \text{ where}$$
$$\rho = (\lambda x. \perp, fix(\lambda r. r[\lambda(y_1, \ldots, y_k). \mathbf{E}[\![E_i]\!]([y_j/x_j], r)/F_i]))$$
$$\mathbf{E}: Exp \to Env \to D$$
$$Env \equiv Varenv \times Funenv$$
$$Varenv \equiv Var \to D$$
$$Funenv \equiv Fun \to D^* \to D$$
$$\mathbf{E}[\![A]\!]\rho = a$$
$$\mathbf{E}[\![x]\!](\rho_1, \rho_2) = \rho_1(x)$$
$$\mathbf{E}[\![B(E_1, \ldots, E_m)]\!]\rho = b(\text{Tuple}(\mathbf{E}[\![E_1]\!]\rho, \ldots, \mathbf{E}[\![E_m]\!]\rho))$$
$$\mathbf{E}[\![F(E_1, \ldots, E_k)]\!](\rho_1, \rho_2) = \rho_2(F)(Tuple(\mathbf{E}[\![E_1]\!](\rho_1, \rho_2), \ldots, \mathbf{E}[\![E_k]\!](\rho_1, \rho_2)))$$

where *Tuple* constructs a tuple of arguments for the function.

In order to make this a complete specification of the semantics we need to specify the domain D and the meanings of Atoms and basic functions (plus,times,if,...). The standard semantics is given by choosing D to be a disjoint union of numbers, booleans and other primitive types and taking the usual meanings for basic functions.

Mycroft follows the Cousots in providing a static semantics which he calls the *collecting* interpretation; he uses the Plotkin powerdomain construction as the basis for this interpretation. Thus the choice for D is the powerdomain of primitive values and the basic functions are lifted in the usual way. For the collecting interpretation we have:

$$\begin{aligned} Values &= Num + Bool + \cdots \\ D &= P(Values) \end{aligned}$$

and for example:

$$\begin{aligned} &+ : P(D^*) \rightarrow P(D) \\ &+(S) = (P+)(S) = \{ +(x,y) \mid (x,y) \in S \}^* \end{aligned}$$

where $\{-\}^*$ is a closure operator that ensures that the set is an element of the Plotkin powerdomain, the details of which need not concern us. Other abstract interpretations are related to the collecting interpretation by a pair of functions (*Abs*, *Conc*). However, the Plotkin powerdomain is not necessarily a complete lattice so that Cousots' framework does not directly apply. With the powerdomain there is an associated subset ordering [Hen78]. There is a continuous operation called union with satisfies the standard axioms and gives us a natural definition of subset:

$$l_1 \subset l_2 \Leftrightarrow l_1 \cup l_2 = l_2$$

An abstract interpretation involves specifying a domain (not necessarily a complete lattice) for D and a set of basic functions. We require *Abs* and *Conc* to obey the following equations:

$$\begin{aligned} Abs \circ Conc &= id \\ Conc \circ Abs &\supset id \end{aligned}$$

Having defined the *Abs* function, the *Conc* function is determined by these properties:

$$Conc(m) = \bigcup \{ l \mid Abs(l) = m \}$$

The correctness theorem of Mycroft shows that

$$g \subset Conc \circ h \circ Abs$$

where g is a function in the collecting interpretation and h is a function in the abstract interpretation.

An example is strictness analysis. This analysis allows us to detect whether a function is strict; a function f is strict if:

$$f \bot = \bot$$

Of course, from the foregoing discussion it is not possible to detect all such information. As stated in the introduction the importance of such an analysis is that the parameters in which a function is strict may be passed by value. In a sequential setting this avoids the need for building a closure and in a parallel setting allows us to evaluate those parameters in parallel with the application.

For strictness analysis we choose D to be the two point domain, 2:

$$\begin{tikzcd} 1 \arrow[d, dash] \\ 0 \end{tikzcd}$$

Intuitively, the 0 is used to represent the undefined element (non-termination) and 1 represents possible termination. The abstraction function is defined in the following way:

$$\begin{aligned} &HALT: P(Values) \rightarrow 2 \\ &HALT(v) = 0, \text{ if } v = \{\perp\} \\ &\qquad\qquad\quad 1, \text{ otherwise} \end{aligned}$$

For strict basic functions we use conjunction:

$$\begin{aligned} &0 \textbf{ and } a = a \textbf{ and } 0 = 0 \qquad a = 0 \quad \text{or} \quad a = 1 \\ &1 \textbf{ and } 1 = 1 \end{aligned}$$

so that

$$+^{\#}(x,y) = x \textbf{ and } y$$

(Mycroft uses the # superscript to represent the strictness analysis interpretation for a function.) For the conditional, Mycroft suggests the following interpretation:

$$if^{\#}(x,y,z) \equiv x \textbf{ and } (y \textbf{ or } z)$$

where **or** has the obvious definition; this captures the intuition that the conditional *must* evaluate its predicate but we cannot know until run-time which of the consequent or alternative is needed (this is where the inaccuracy creeps into our interpretation).

Thus deviating slightly from the original syntax, the abstract interpretation for the factorial function:

$$fac(x) = if(=(x,0), *(x, fac(-(x,1))))$$

is given by

$$fac^{\#}(x) = (x \textbf{ and } 1) \textbf{ and } (1 \textbf{ or } x \textbf{ and } fac^{\#}(x \textbf{ and } 1))$$

(notice that atoms are definitely defined and thus get mapped to 1 by the abstract interpretation). After simplification we get:

$$fac^{\#}(x) = x \textbf{ and } fac^{\#}(x)$$

and we can give meaning to this by the usual fixed point techniques (the first iteration being $\lambda x.0$). The fixed point gives:

$$fac^{\#}(x) = 0$$

and we can use this to infer that *fac* is strict since

$$fac^{\#}(0) = 0 \qquad (\text{i.e. } fac \perp = \perp)$$

The Burn, Hankin and Abramsky framework described later in this chapter is an extension of this approach to higher-order languages and Wadler's chapter addresses the problem of analysing structured data (lists) within this framework.

Mycroft shows how his framework may be applied to other optimizations of functional languages. The major example is in-place update analysis. This is a rather complex analysis that involves three different interpretations; an alternative, more recent approach is reported in Hudak's chapter.

5. NIELSON – A GENERAL FRAMEWORK

The most general framework to date has been presented by Flemming Nielson in his thesis and developments therefrom. A lucid account of this work is given in Nielson's chapter and we shall not rehearse it here. He and his co-workers have used the framework for a variety of traditional flow analysis problems and, more recently, strictness analysis and in-place update analysis for functional programs.

6. BURN, HANKIN AND ABRAMSKY – ABSTRACT INTERPRETATION IN THE APPLICATIVE IDIOM II

Mycroft's pioneering work on strictness analysis in his thesis was confined to the case of first-order functions over flat domains of basic data. In order to make strictness analysis really applicable to practical functional programming, two extensions were essential:

(1) To non-flat domains, e.g. lazy lists, trees etc. This is the topic of the Chapters 4 and 12, and is also mentioned as an application of his methods by Jones.
(2) To higher-order (and preferably polymorphic) functions. This topic was addressed in a number of papers which appeared at about the same time [Bur86b], [Hud86b], [Mau86] and [Wra85]. Since [Bur86b] is referenced by a number of chapters in this volume, we shall give a summary of the main ideas here.

6.1 A Syntax for Higher-order Functions

We shall take the typed λ-calculus as a simple system in which the essential ideas can be expressed. Firstly, we have type expressions, given by the syntax:

$$\sigma := A \mid \sigma \rightarrow \tau$$

where A ranges over some collection of base types.

Examples If **int**, **bool** are base types, then

$$(\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{bool}$$
$$(\mathbf{int} \rightarrow \mathbf{int}) \rightarrow (\mathbf{int} \rightarrow \mathbf{int})$$

are type expressions.

Next, we have typed terms (i.e. 'programs') described as follows. For each type σ we have a set of variables $Var_\sigma = \{x^\sigma, y^\sigma, z^\sigma, \ldots\}$. We also have a collection of typed *constants*, c_σ. Then we form terms M of type σ (written

$M{:}\sigma$) according to the following rules:

(1) $x^\sigma{:}\sigma$
(2) $c_\sigma{:}\sigma$
(3) $M{:}\sigma \to \tau \quad N{:}\sigma \Rightarrow MN{:}\tau$
(4) $M{:}\tau \Rightarrow \lambda x^\sigma.M{:}\sigma \to \tau$

We want to consider both *standard* and *abstract* interpretations of the calculus. The general concept of which these are both instances is *interpretation*. An interpretation I assigns a domain D^I_A to each base type A. This is then extended to all types σ by:

$$D^I_{\sigma\to\tau} \equiv [D^I_\sigma \to D^I_\tau]$$

Moreover I assigns a denotation $c^I_\sigma \in D^I_\sigma$ to each constant c_σ of type σ. This is then extended to all terms by the *semantic function* $[\![M]\!]^I\rho$, which maps a term $M{:}\sigma$ and an *environment* ρ to an element of D^I_σ. (An environment is a map from variables x^σ to elements of D^I_σ). $[\![\;]\!]^I$ is defined as follows:

$$\begin{aligned}
&[\![x^\sigma]\!]^I\rho = \rho x^\sigma\\
&[\![c_\sigma]\!]^I\rho = c^I_\sigma\\
&[\![MN]\!]^I\rho = ([\![M]\!]^I\rho)\ ([\![N]\!]^I\rho)\\
&[\![\lambda x^\sigma.M]\!]^I\rho = \lambda d \in D^I_\sigma.[\![M]\!]^I\rho[d/x^\sigma]
\end{aligned}$$

Thus an interpretation is determined by the domains it assigns to base types, and the values it assigns to constants.

Example Suppose the base types are **int** and **bool**, and the constants are:

$$\begin{aligned}
&0{:}\mathbf{int}\\
&tt, f\!f{:}\mathbf{bool}\\
&succ, pred{:}\mathbf{int} \to \mathbf{int}\\
&zero{:}\mathbf{int} \to \mathbf{bool}\\
&\text{for each } \sigma,\ if_\sigma{:}\mathbf{bool} \to \sigma \to \sigma \to \sigma
\end{aligned}$$

The standard interpretation St would be:

$$\begin{aligned}
&D^{St}_{\mathbf{int}} = N_\perp\\
&D^{St}_{\mathbf{bool}} = Bool_\perp\\
&0^{St} = 0\\
&succ^{St}\perp = \perp\\
&succ^{St} n = n + 1\\
&if^{St}_\sigma \perp x\, y = \perp\\
&if^{St}_\sigma\, tt\, x\, y = x\\
&if^{St}_\sigma\, f\!f\, x\, y = y
\end{aligned}$$

etc. Now an abstract interpretation will be just another interpretation *Abs*. The abstract interpretation for strictness analysis will be the obvious extension of Mycroft's. For our example, this is as follows (we have followed Mycroft in calling this interpretation #):

$$D^{\#}_{\mathbf{int}} = D^{\#}_{\mathbf{bool}} = 2, \text{ the two-point domain}$$
$$succ^{\#} = pred^{\#} = zero^{\#} = \text{the identity function on } 2$$

More generally, if $f: A_1 \to \cdots \to A_n \to A$ is a (curried) first-order function of n arguments, and strict in each argument, then:

$$f^{\#}: 2 \to \cdots \to 2 \to 2$$

is an n-ary conjunction:

$$f^{\#}(b_1, \ldots, b_n) = b_1 \ \mathbf{and} \ldots \mathbf{and} \ b_n$$

Finally, the conditional:

$$if^{\#}_{\sigma}\, 0\, x\, y = 0$$
$$if^{\#}_{\sigma}\, 1\, x\, y = x \sqcup y$$

where $x \sqcup y$ is the least upper bound (computed pointwise). This always exists, since each $D^{\#}_{\sigma}$ is a lattice. In the case when σ is a ground type, this yields Mycroft's original interpretation of the conditional.

This then is our abstract interpretation. To see how we may actually use it to obtain strictness information, consider a term of the form:

$$M: \sigma_1 \to \cdots \to \sigma_n \to A$$

To test if this term (denotes a function which) is strict in the ith argument, we can test:

$$[\![M]\!]^{\#} \top^{\#}_{\sigma 1} \cdots \bot^{\#}_{\sigma_i} \top^{\#}_{\sigma_n} = 0 \qquad (*)$$

where $\top^{\#}_{\sigma}, \bot^{\#}_{\sigma}$ are the top and bottom elements respectively in the domain $D^{\#}_{\sigma}$. (For simplicity, we assume that M is closed and ignore the environment). Since all the objects and operations in the abstract domain are finite and can be explicitly tabulated, (*) is an effective test, which can be implemented in a compiler. For further discussion of how this abstract interpretation can be used to guide efficient compilation, see [Han86].

The question remains, however, whether (*) is actually *correct*; that is, whether we can soundly infer from (*) being satisfied that M is indeed strict in its ith argument in the *standard* semantics, and therefore in its actual computational behaviour. To guarantee that this is so, we obviously must relate the standard and abstract interpretations in such a way as to guarantee that properties inferred by calculations in the abstract domain are satisfied in the standard domains. The main thrust of [Bur86b] is to establish this relationship.

The idea is to define *abstraction functions*

$$abs_{\sigma}: D^{St}_{\sigma} \to D^{\#}_{\sigma}$$

which satisfy some crucial properties:

(1) *abs* preserves $\bot$:
$$abs_{\sigma} \bot^{St}_{\sigma} = \bot^{\#}_{\sigma}$$

(2) *abs* reflects $\perp$:
$abs_\sigma d = \perp^\#_\sigma \Rightarrow d = \perp^{St}_\sigma$

(3) *abs* is a semi-homomorphism of semantics:
For each $M:\sigma$, and environment ρ^{St},

$$abs_\sigma(\llbracket M \rrbracket^{St}\rho) \sqsubseteq \llbracket M \rrbracket^\#(abs \circ \rho)$$

(4) *abs* is a semi-homomorphism of application:
$abs_\tau(f\,d) \sqsubseteq abs_{\sigma\to\tau}(f)\,abs_\sigma(d)$

From (1)–(4), the main result of [Bur86b] can easily be proved:

Soundness Theorem For Strictness Analysis

For $M:\sigma_1 \to \cdots \to \sigma_n \to A$,

$$\llbracket M \rrbracket^\# \top^\#_{\sigma_1} \cdots \perp^\#_{\sigma_i} \cdots \top^\#_{\sigma_n} = 0 \Rightarrow \text{For all } d_1 \in D^{St}_{\sigma_1}, \ldots, d_n \in D^{St}_{\sigma}:$$
$$\llbracket M \rrbracket^{St} d_1 \cdots \perp^{St}_{\sigma_i} \cdots d_n = \perp^{St}_A$$

In other words, the inference back from our test to actual computational behaviour is valid.

Proof of the Soundness Theorem

$\llbracket M \rrbracket^\# \top^\#_{\sigma_1} \cdots \perp^\#_{\sigma_i} \cdots \top^\#_{\sigma_n} = 0 \Rightarrow abs\ \llbracket M \rrbracket^{St} abs_\sigma(d_1) \ldots abs_{\sigma_i}(\perp^{St}_{\sigma_i}) \ldots abs_{\sigma_n}(d_n) = 0$
by (1), (3) and monotonicity

$\Rightarrow abs_A(\llbracket M \rrbracket^{St} d_1 \cdots \perp^{St}_{\sigma_i} \cdots d_n) = 0$ by (4)

$\Rightarrow \llbracket M \rrbracket^{St} d_1 \cdots \perp^{St}_{\sigma_i} \cdots d_n = \perp^{St}_A$ by (2) ■

How can we define abstraction functions satisfying (1)–(4) above? The cases for base types are easy – they just follow Mycroft's definition of his *HALT* function:

$$abs_A d = 0 \text{ if } d = \perp^{St}_A$$
$$1, \text{ otherwise.}$$

The extension to higher-order types is less obvious. In [Bur86b] it is done using the formalism of powerdomains. [Abr85b] gives an equivalent but simpler definition;

Definition

$$abs_{\sigma\to\tau} f b = \bigsqcup \{(abs_\tau \circ f)d \mid abs_\sigma d \sqsubseteq b\}.$$

Intuitively, given $f: D^{St}_{\sigma\to\tau}$ and $b \in D^\#_\sigma$, we must consider all those $d \in D^{St}_\sigma$ which are *safely related to* b, i.e. for which $abs_\sigma d \sqsubseteq b$. We then consider the set of all abstractions of images of such d under f. The result of our abstraction of f applied to b must be safely related to everything in this set – must be an upper bound – but should lose as little information as possible in doing so – hence the *least* upper bound.

The technical development in [Bur86b] is concerned with showing that these functions have the required properties. [Abr85b] puts these ideas in a wider context relating them to standard notions in the λ-calculus (logical relations) and category theory (Kan extensions). The extension of the ideas of [Bur86b] to polymorphic functions is investigated in [Abr86].

CONCLUSIONS

We have presented a selection of the major developments in abstract interpretation that form the background for the other chapters in the book. Mycroft's Jones' and Foster's chapters present alternative approaches to the subject.

This chapter would not be complete without some mention of the people and organizations that have assisted us in our work on abstract interpretation. Firstly, we must thank Geoff Burn who has collaborated with us for the last two years and who first introduced us to these ideas through his thesis work. Secondly, we must thank everyone who attended our workshop at the University of Kent in 1985, the Alvey Directorate who funded that event and Richard Sykes who organized it.

Bibliography

This bibliography is primarily compiled from references cited in the text. Some additional references have been added from a bibliography on abstract interpretation which has been prepared by Flemming Nielson. Abstract interpretation is closely related to flow analysis; [Muc81] contains an excellent bibliography of this area.

ABBREVIATIONS

ACM	Association of Computing Machinery
CACM	Communications of the ACM
ICALP	International Colloquium on Automata, Languages and Programming
IFIP	International Federation of Information Processing
JACM	Journal of the ACM
JIMA	Journal of the Institute of Mathematical Applications
LNCS	Lecture Notes in Computer Science
MFCS	Symposium on Mathematical Foundations of Computer Science
POPL	ACM Symposium on Principles of Programming Languages
TOPLAS	ACM Transactions on Programming Languages and Systems

[Abr85a] Abramsky S. and Sykes R. SECD-M: A virtual machine for applicative programming, *Proceedings IFIP Symposium on Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, 1985

[Abr85b] Abramsky S. *Abstract Interpretation, logical relations and Kan extensions*, unpublished manuscript, 1985

[Abr86] Abramsky S. Strictness Analysis and polymorphic invariance, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Aho77] Aho A. V. and Ullman J. D. *Principles of compiler design*, Addison-Wesley, 1977

[All76] Allen F. E. and Cocke J. A program data flow analysis procedure, *CACM* **19**(3), 137–147, 1976

[And87] Anderson N. *Approximating term rewriting systems by regular tree grammars*, forthcoming, 1987

[Aug84] Augustsson L. A compiler for lazy ML, *Proceedings of the ACM Symposium on Lisp and Functional Programming*, 1984

[Bac75] Backhouse R. C. and Carre B. A. Regular algebra applied to path-finding problems, *JIMA* **15**, 161–186, 1975

[Bac78] Backus J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *CACM* **21**(8), 1978

[Bar77] Barth J. M. Shifting garbage collection overhead to compile time, *CACM* **20**(7), 513–518, 1977

[Bar83] Barbuti R. and Martelli A. A structural approach to static semantics correctness. *Science of Computer Programming* 3, 279–311, 1983

[Bar84] Barendregt H. P. *The Lambda Calculus – Its Syntax and Semantics*, North-Holland. 1984

[Bau85] Bauer F. L. *et al The Munich Project CIP*–Vol 1: *The wide spectrum CIP-L*, LNCS 183, Springer-Verlag, 1985

[Bra69] Brainerd W. S. Tree generating regular systems, *Information and Control* **13**, 217–231, 1969

[Bra82a] Bramson B. D. and Goodenough S. J. *Data use analysis for computer programs*, unpublished R.S.R.E. report, 1982

[Bra82b] Bramson B. D. *Information flow analysis for computer programs*, unpublished R.S.R.E. report, 1982

[Bry85] Bryant R. E. *Graph-based algorithms for boolean function manipulation*, CMU-CS-85-135, Department of Computer Science, Carnegie-Mellon University, 1985

[Bur80] Burstall R., MacQueen D. and Sannella D. HOPE: An Experimental Applicative Language, *Proceedings of the ACM Conference on Lisp and Functional Languages*, 1980

[Bur85] Burn G. L. *Why the problem of polymorphism and strictness analysis has not been solved*, Note on the FP Bulletin Board, 1985

[Bur86a] Burn G. L., Hankin C. L. and Abramsky S. The theory of strictness analysis for higher order functions, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Bur86b] Burn G. L., Hankin C. L. and Abramsky S. Strictness analysis for higher order functions, *Science of Computer Programming* 7, 249–278, 1986

[Bur87] Burn G. L. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*, PhD thesis, University of London, 1987

[Car79] Carre B. A. *Graphs and networks*, Oxford University Press, 1979

[Cla77] Clark D. W. An empirical study of list structure in Lisp, *CACM* **20**(2), 78–87, 1977

[Cla85] Clack C. D. and Peyton Jones S. L. Strictness Analysis – a practical approach, *Proceeding IFIP Symposium on Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, 1985

[Cle86] Clement D., Despeyroux J., Despeyroux T. and Kahn G. A simple applicative language: Mini-ML, *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1986

[Cou76] Cousot P. and Cousot R. Static determination of dynamic properties of programs, *Proceedings of the 2nd International Symposium on Programming*, 1976

[Cou77a] Cousot P. and Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints, *4th POPL*, pp 238–252, 1977

[Cou77b] Cousot P. and Cousot R. Static determination of dynamic properties of generalised type unions, *ACM SIGPLAN Notices* **12**(3), 77–94, 1977

[Cou77c] Cousot P. and Cousot R. Automatic synthesis of optimal invariant assertions mathematical foundations, *ACM SIGPLAN Notices* **12**(8), 1977

[Cou78a] Cousot P. and Cousot R. Static Determination of Dynamic Properties of Recursive Procedures, in *Formal Descriptions of Programming Concepts*, edited by Neuhold E. J., North-Holland, 1978

[Cou78b] Cousot P. and Halbwachs N. Automatic discovery of linear restraints among variables of a program, *5th POPL*, 1978

[Cou78c] Cousot P. *Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotone sur un treillis analyse sémantique des programmes*, Thèse Docteur des Sciences Mathématiques, Grenoble, France, 1978

[Cou79] Cousot P. and Cousot R. Systematic design of program analysis frameworks, *6th POPL*, 1979

[Cou80] Cousot P. and Cousot R. Semantic Analysis of Communicating Sequential Processes, *ICALP 1980*, LNCS 85, Springer-Verlag, 1980

[Cou81] Cousot P. Semantic foundations of program analysis, in *Program Flow Analysis: Theory and Applications*, edited by Muchnick S. S. and Jones N. D., Prentice-Hall, 1981

[Cur81] Currie I. F., Edwards P. W. and Foster J. W. *Flex firmware*, R.S.R.E. Report 81009, 1981

[Cur85a] Curien P-L. *Categorical Combinators, Sequential Algorithms and Functional Programming*, CNRS-Université Paris VII, LITP, 1985

[Cur85b] Curtis K., Hamond P. D. and Taylor P. D. *Flex Pascal: an implementation of the ISO-Pascal programming language*, R.S.R.E. Memorandum 3908, 1985

[Dar81] Darlington J. and Reeve M. Alice – A Multi-processor reduction machine for the parallel evaluation of applicative languages, *Proceedings of the ACM Conference on Functional Languages and Computer Architecture*, New Hampshire, 1981

[Deb85] Debray S. *Automatic Mode Inference for Prolog programs*, Report #85/019, Department of Computer Science, SUNY at Stony Brook, New York, 1985

[Deb86] Debray S. *Dataflow analysis of logic programs*, Draft report, Department of Computer Science, SUNY at Stony Brook, New York, 1986

[Dij76] Dijkstra E. *A Discipline of Programming*, Prentice-Hall, 1976

[Don78] Donzeau-Gouge V. *Utilisation de la sémantique dénotationelle pour l'étude d'interpretations non-standard*, INRIA rapport 273, 1978

[Fai82] Fairbairn J. *Ponder and its type system*, Technical Report 31, University of Cambridge Computer Laboratory, 1982

[Fai85] Fairbairn J. Removing redundant laziness from Super-combinators, *Proceedings of the Workshop on Implementation of Functional Languages*, Aspenaes, Sweden, 1985

[Fos76] Fosdick L. D. and Osterweil L. J. Data flow analysis in software reliability, *ACM Computing Surveys* 8(3), 305–330, 1976

[Fos85] Foster J. M. *Regular expression analysis of procedures and exceptions*, R.S.R.E. Report 85008, 1985

[Fos86] Foster J. M. Validating microcode algebraically, *Computer Journal*, 29(5), 416–442, 1986

[Fri77] Friedman D. and Wise D. S. Aspects of applicative programming for file systems, *Proceedings of the ACM Conference on Language Design for Reliable Software, ACM SIGPLAN Notices* 12(3), 1977

[Gan86] Ganzinger H. and Jones N. D. (eds) *Program as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Géc84] Gécseg F. and Steinby M. *Tree Automata*, Akademiai Kiado, Budapest, 1984

[Gie81] Giegerich R., Möncke U. and Wilhelm R. *Invariance of Approximate Semantics with respect to Program Transformations*, Springer IFP 50, 1981

[Gie83] Giegerich R. A Formal Framework for the Derivation of Machine-Specific Optimizers, *TOPLAS* 5(3), 1983

[Gog76] Goguen J. A., Thatcher J. W., Wagner E. G. and Wright J. B. Some fundamentals of order algebra semantics, *Proceedings of MFCS 1976*, LNCS 45, Springer-Verlag, 1976

[Gog77] Goguen J. A., Thatcher J. W., Wagner E. G. and Wright J. B. Initial algebra semantics and continuous algebras, *JACM* 24(1), 1977

[Goo86] Goodenough S. J. Ten15 and the RSRE Ada Flex compiler, *Ada User*, 1986

[Gor79a] Gordon M., Milner R. and Wadsworth C. *Edinburgh LCF*, LNCS 78, Springer-Verlag, 1979

[Gor79b] Gordon M. J. C. *The Denotational Description of Programming Languages*, Springer-Verlag, 1979

[Han86] Hankin C. L., Burn G. L. and Peyton Jones S. L. A safe approach to parallel combinator reduction, *Proceedings of the European Symposium on Programming*, LNCS 213, Springer-Verlag, 1986

[Hec77] Hecht M. *Flow Analysis of Computer Programs*, North-Holland, 1977

[Hen78] Hennessey M. and Plotkin G. Full Abstraction for a simple parallel programming language, *Proceedings of MFCS 1978*, LNCS 64, Springer-Verlag, 1978

[Hen82] Henderson P. Purely Functional Operating Systems, in *Functional Programming and its Applications*, Cambridge University Press, 1982

[Hin69] Hindley R. The principal type-scheme of an object in combinatory logic, *Transactions of the American Mathematical Society* 146(1), 29–60, 1969

[Hoa62] Hoare C. A. R. Quicksort, *Computing Journal* 5(4), 10–15, 1962

[Hud84] Hudak P. and Kranz D. A Combinator-based Compiler for a Functional Language, *11th POPL*, 1984

[Hud85a] Hudak P. and Bloss A. The aggregate update problem in functional programming systems, *12th POPL*, 300–314, 1985

[Hud85b] Hudak P. and Young J. *A set-theoretic characterisation of function strictness in the lambda calculus*, Technical Report YALEU/DCS/RR-391, Yale University, 1985

[Hud86a] Hudak P. *Collecting interpretations of expressions*, Research report 497, Yale University, Department of Computer Science, 1986

[Hud86b] Hudak P. and Young J. Higher-order Strictness Analysis in untyped lambda calculus, *13th POPL*, 1986

[Hue79] Huet G. and Lévy J. J. *Call by need computations in nonambiguous linear term rewriting systems*, Report 359, INRIA, France, 1979

[Hug84] Hughes J. *Why functional programming matters*, Programming Methodology Group Report 16, CTH Gothenburg, Sweden, 1984

[Hug86] Hughes J. Strictness Detection in Non-flat Domains, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Joh84] Johnsson T. Efficient compilation of lazy evaluation, *Proceedings of the ACM/SIGPLAN Notices Conference on Compiler Construction*, 1984

[Joh85] Johnsson T. Lambda lifting: transforming programs to recursive equations, *Proceedings IFIP Symposium on Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, 1985

[Jon79a] Jones N. D. and Muchnick S. S. Flow analysis and optimisation of LISP-like structures, *6th POPL*, 244–256, 1979

[Jon79b] Jones N. D. and Muchnick S. S. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra, *Proceedings of the 20th Conference on Foundations of Computer Science*, 1979

[Jon81] Jones N. D. Flow analysis of lambda expressions, *ICALP 1981*, LNCS 115, Springer-Verlag, 1981

[Jon82] Jones N. D. and Muchnick S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures, *9th POPL*, 66–74, 1982

[Jon84] Jones S. B. *A range of purely functional Operating Systems*, Stirling University, Scotland, 1984

[Jon85a] Jones N. D., Sestoft P. and Søndergaard H. An experiment in partial evaluation: the generation of a compiler generator, in *Rewriting Techniques and Applications*, LNCS 202, Springer-Verlag, 1985

[Jon85b] Jones N. D. *Concerning the abstract interpretation of Prolog*, draft paper, DIKU, Copenhagen, 1985

[Jon86] Jones N. D. and Mycroft A. Data flow analysis of applicative programs using minimal function graphs, *13th POPL*, 1986

[Jou85] Jouannaud J. P. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, 1985

[Kam77] Kam J. B. and Ullman J. D. Monotone data flow analysis frameworks, *Acta Informatica* 7, 305–317, 1977

[Kar85] Karlsson K. *Strictness analysis*, talk delivered at the Workshop on Abstract Interpretation, University of Kent, 1985

[Kar86] Karp R. M. Combinatorics, complexity and randomness, *CACM* 29(2), 98–111, 1986

[Kel81] Keller R. M., Lindstrom G. and Patil S. A loosely coupled applicative multiprocessing system, *AFIPS*, 1979

[Kie83] Kieburtz R. Precise typing of abstract data type specifications, *10th POPL*, 1983

[Kie85] Kieburtz R. *Strictness detection in non-flat domaiins*, talk delivered at Oxford University, 1985

[Kor79] Kornfeld W. *Combinatorially implosive algorithms*, Computer Laboratory, MIT, 1986

[Llo84] Lloyd J. W. *Foundations of Logic Programming*, Springer-Verlag, 1984

[Mac82] MacQueen D. B. and Sethi R. A semantic model of types for applicative languages, *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1982

[Mau86] Maurer D. Strictness Computations Using Generalised λ-expressions, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Mei84] Meira S. Optimised combinatoric code for applicative language implementation, *Proceedings of the 6th International Symposium on Programming*, Springer-Verlag, 1984

[Mel81] Mellish C. S. *The Automatic Generation of Mode Declarations for Prolog Programs*, DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, 1981

[Mel85] Mellish C. S. Some global optimisations for a Prolog compiler, *Journal of Logic Programming* 2(1), 43–66, 1985

[Mel86] Melton A., Schmidt D. A. and Strecker G. E. Galois connections and computer science applications, *Proceedings of the Workshop on Category Theory and Computer Programming*, LNCS 240, Springer-Verlag, 1986

[Mez67] Mezei J. and Wright J. B. Algebraic automata and context-free sets, *Information and Control* 11, 3–29, 1967

[Mil76a] Milner R. E. and Strachey C. *A Theory of Programming Language Semantics*, Chapman and Hall, 1976

[Mil76b] Milner R. Program Semantics and Mechanised Proof, *Mathematical Centre Tracts* **82**, Amsterdam, 1976

[Mil76c] Milner R. Models of LCF, *Mathematical Centre Tracts* **82**, Amsterdam 1976

[Mil78] Milner R. A theory of type polymorphism in programming, *Journal of Computer and System Sciences* 17, 348–375, 1978

[Mil80] Milner R. *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag, 1980

[Mis84] Mishra P. and Keller R. M. Static inference of properties of applicative programs, *11th POPL*, 1984

[Mis85] Mishra P. *Static inference in applicative languages*, PhD thesis, University of Utah, 1985

[Mos81] Mosses P. D. *A semantic algebra for binding constructs*, DIAMI report PB-132, Department of Computer Science, Aarhus University, 1981

[Mos82] Mosses P. D. Abstract semantic algebras, *Proceedings of IFIP TC2 Working Conference on formal description of programming concepts II*, North-Holland, 1982

[Muc81] Muchnick S. S. and Jones N. D. (eds) *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981

[Myc80] Mycroft A. The theory and practice of transforming call-by-need into call-by-value, *Proceedings of the 4th International Symposium on Programming*, LNCS 83, Springer-Verlag, 1980

[Myc81] Mycroft A. *Abstract interpretation and optimising transformations for applicative programs*, PhD thesis, University of Edinburgh, 1981

[Myc83] Mycroft A. and Nielson F. Strong abstract interpretation using power domains, *ICALP 1983*, LNCS 154, Springer-Verlag, 1983

[Myc86] Mycroft A. and Jones N. D. A relational framework for abstract interpretation, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Nie82] Nielson F. A denotational framework for data flow analysis, *Acta Informatica* **18** 265–287, 1982

[Nie84] Nielson F. *Abstract interpretation using domain theory*, PhD thesis, University of Edinburgh, 1984

[Nie85a] Nielson F. Program transformations in a denotational setting, *TOPLAS* 7(3), 359–379, 1985

[Nie85b] Nielson F. Tensor products generalize the relational data flow analysis method, *Proceedings of the 4th Hungarian Computer Science Conference*, 1985

[Nie86a] Nielson F. Abstract interpretation of denotational definitions, *Proceedings STACS 1986*, LNCS 210, 1986

[Nie86b] Nielson F. Expected forms of data flow analysis, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Nie86c] Nielson F. *Strictness analysis and denotational abstract interpretation*, unpublished manuscript, 1986

[Nie86d] Nielson H. R. and Nielson F. Pragmatic aspects of two-level denotational meta-languages, *Proceedings of the European Symposium on Programming*, LNCS 213, Springer-Verlag, 1986

[Nie86e] Nielson H. R. and Nielson F. Code generation from two-level denotational meta-languages, *Proceedings of the DIKU Workshop on Programs as Data Objects*, LNCS 217, Springer-Verlag, 1986

[Nie86f] Nielson H. R. and Nielson F. *A tutorial on TML, the meta-language of the PSI project*, Aalborg University, Denmark, 1986

[Nie86g] Nielson H. R. and Nielson F. Semantics Directed Compiling for Functional Languages, *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1986

[Pan84a] Panangaden P. and Mishra P. *A category theoretic formalism for abstract interpretation*, Manuscript, 1984

[Pan84b] Panangaden P. and Mishra P. *Abstraction and indeterminacy*, Technical Report UUCS-84-006, University of Utah, 1984

[Pey86] Peyton Jones S. L. Functional programming languages as a software engineering tool, in *Software Engineering – the Critical Decade*, Peter Peregrinus, 1986

[Pey87] Peyton Jones S. L. *Implementing Functional Languages using Graph Reduction*, Prentice-Hall, 1987

[Pla84] Plaisted D. The occur-check problem in Prolog, *Proceedings of the International Symposium on Logic Programming*, New Jersey, 1984

[Plo80] Plotkin G. Lambda definability in the full type hierarchy, in [Sel80]

[Rao84] Raoult J.-C. and Sethi R. The global storage needs of a subcomputation, *11th POPL*, 148–157, 1984

[Rey69] Reynolds J. Automatic computation of data set definitions, *Information Processing* **68**, 456–461, North-Holland, 1969

[Rey74] Reynolds J. On the relation of direct and continuation semantics, *ICALP 1974*, LNCS 14, Springer-Verlag, 1974

[Rey83] Reynolds J. Types, abstraction and parametric polymorphism, *IFIP '83*, North-Holland, 1983

[Ros80] Rosen B. K. Monoids for rapid data flow analysis, *SIAM Journal of Computing* **9**, 159–196, 1980

[Sal66] Salomaa A. Two complete axiom systems for the algebra of regular events, *JACM* **13**(1), 158–169, 1966

[Sch78] Schwarz J. Verifying the safe use of destructive operations in applicative programs, in *Program Transformations – Proceedings of the 3rd International Symposium on Programming*, edited by Robinet B., Dunod Informatique, 1978

[Sch85] Schmidt D. A. Detecting global variables in denotational specifications, *TOPLAS* **7**(2), 299–310, 1985

[Sch86] Schmidt D. A. *Denotational Semantics*, Allyn and Bacon, 1986

[Sco76] Scott D. S. Data types as lattices, *SIAM Journal of Computing* **5**(3), 522–587, 1976

[Sco82] Scott D. S. Domains for denotational semantics, *ICALP 1982*, LNCS 140, Springer-Verlag, 1982

[Sel80] Seldin J. P. and Hindley J. R. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980

[Scha77] Shamir A. and Wadge W. W. Data types as objects, *ICALP 1977*, LNCS 52, 1977

[Sic76] Sickel S. A search technique for clause interconnectivity graphs, *IEEE Transactions on Computers* **25**(8), 1976

[Sin72] Sintzoff M. Calculating properties of programs by valuation on specific models, Proceedings of the ACM Conference on Proving Assertion about Programs, *ACM SIGPLAN Notices* 7(1), 203–207, 1972

[Smy82] Smyth M. B. and Plotkin G. D. The category-theoretic solution of recursive domain equations, *SIAM Journal of Computing* 11(4), 1982

[Søn86] Søndergaard H. An application of abstract interpretation of logic programs: Occur check reduction, *Proceedings of the European Symposium on Programming*, LNCS 213, Springer-Verlag, 1986

[Sto77] Stoy J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977

[Tar81] Tarjan R. E. A unified approach to path programs, *JACM* 28(3), 577–593, 1981

[Tar55] Tarski A. A lattice-theoretical fixpoint theorem and its applications, *Pacific Journal of Mathematics* 5, 285–309, 1955

[Ten74] Tenenbaum A. *Automatic types analysis in a very high level language*, PhD thesis, New York University, 1974

[Ten81] Tennent R. D. *Principles of Programming Languages*, Prentice-Hall, 1981

[Tha73] Thatcher J. Tree automata: an informal survey, in *Currents in the theory of Computing*, edited by Aho A. V., Prentice-Hall, 1973

[Tur80] Turchin V. *The language REFAL, the theory of compilation, and metasystem analysis*, Courant Institute Report 20, New York, 1980

[Tur81] Turner D. A. The semantic elegance of applicative languages, *Proceedings of the ACM Symposium on Functional Languages and Computer Architecture*, New Hampshire, 1981

[Tur85] Turner D. A. Miranda: a non-strict functional language with polymorphic types, *Proceedings IFIP Symposium on Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, 1985

[Wad85] Wadler P. *An introduction to Orwell*, Programming Research Group, Oxford University, 1985

[Weg75] Wegbreit B. Property extraction in well-founded property sets, *IEEE Transactions on Software Engineering* 1, 270–285, 1975

[Wra85] Wray S. A new strictness detection algorithm, *Proceedings of the Workshop on Implementation of Functional Languages*, Aspenaes, Sweden, 1985