Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Program Analysis Tools

Chris Hankin

Dept. of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

Received: date / Revised version: date

Abstract. This paper introduces a special section on program analysis tools. We start by defining what we mean by "program analysis". We identify and briefly describe three approaches to program analysis. We then introduce the three contributed papers of the section.

1 Introduction

Program analysis is an automatic technique for finding out properties of programs without having to run them. Why do we need to do that? We will just mention two situations for now but we will return to this later:

- suppose we buy some third party software, then we might want to know that it behaves in an acceptable way before integrating it into our live system; for example, we might like to know that it does not generate uncaught exceptions.
- if we can analyse a program and discover that different iterations of a loop are independent, then the loop can be parallelised.

Researchers have been developing program analysis techniques to answer these sorts of questions since the 1960s.

The forthcoming book [12] contains a detailed treatment of the different approaches to program analysis. This special section introduces three tools for program analysis. In this introductory paper we give an informal tutorial for the techniques employed in the other papers in this special section.

1.1 Program Analysis

Program analysis is often also called *static analysis*. As we have already said, the aim of program analysis is to find out something about a program's behaviour without having to run the program. For example, consider the following simple program:

```
1: input n;
2: m := 2;
while 3: n > 1 do
    4: m := m × n;
    5: n := n − 1;
6: output m;
```

where we have preceded each statement by a label for ease of reference. We can statically determine that the value of m at statement 6 will be even for any input n. A program analysis can determine this by propagating parity information forwards from the start of the program. We can assign one of three properties to each variable:

- *even* – the value is known to be even
- *odd* – the value is known to be odd
- *unknown* – the parity of the value is unknown

Taking proper care of the loop (see below), program analysis would determine the following information at the beginning of each statement:

1:	m: *unknown*	n: *unknown*
2:	m: *unknown*	n: *unknown*
3:	m: *even*	n: *unknown*
4:	m: *even*	n: *unknown*
5:	m: *even*	n: *unknown*
6:	m: *even*	n: *unknown*

The program computes 2 times the factorial of n for any positive value of n. Replacing statement 2 by:

```
2: m := 1;
```

gives a program that computes factorials but then the program analysis is unable to tell us anything about the parity of m at statement 6. This is correct because m could be *even* (if the input is greater than 1) or *odd* (if the input is less than or equal to 1). However, even if we fix the input to be positive and even, by some suitable conditional assignment, the program analysis will still not accurately predict the evenness of `m` at statement `6`. This loss of accuracy is a common feature of program analyses: many properties that we are interested in are essentially undecidable and therefore we cannot hope to detect them accurately. We have to ensure that the answers from program analysis are at least *safe*. In this context "safety" has a technical meaning: in general, the program analysis will answer yes or no depending on whether the program has the property that we are seeking; we would prefer that the analysis answered no sometimes when the correct (undecidable) answer is yes rather than vice versa. So in a safe program analysis:

– **yes** means definitely yes, and
– **no** means possibly no.

In the modified factorial program, it is safe to say that the parity of `m` is *unknown* at `6` – it would not be safe to say that `m` is *even*.

In general, information gathered in this way may be used for a variety of different purposes:

Optimising compilers: program analysis can be used in the semantic analysis phase to gather information that can be used to optimise the code that is generated; this is the most developed area and is represented by two of the papers in this section [3,10].

Verification: program analysis can be used to generate information that may be used to verify the correct behaviour of a program. Some early papers on abstract interpretation considered the automatic generation of assertions. The paper [2] in this section addresses verification of concurrent programs.

Formal testing and debugging: program analysis can be used to identify boundary cases that can be used in the automatic generation of test data.

Researchers have worked on program anlaysis techniques since the 1960s and there is, by now, an extensive literature. A more recent area of work has been *semantics-based* program analysis and this is the emphasis of the papers in this section.

The semantics of a program concerns its *meaning*. Formal semantics specify the meaning of a program by mapping the program into some formal meta-language. Two styles of formal semantics are commonly used in program analysis: *denotational* semantics and *operational* semantics. In the former case, the meaning of a program is captured as a continuous function on domains (one may consider a domain to be a set with some additional structure – the details need not concern us). In the latter case, the meaning of programs is captured by their action in some form of (more or less) abstract machine. In semantics-based program analysis the analysis is derived from a formal semantics of the program. The benefit is that we can have confidence that the results of the analysis do correctly reflect an appropriate aspect of the program's dynamic behaviour.

We identify three facets of program analysis:

– specification,
– efficient implementations, and
– correctness

The specification defines which properties the analysis detects and how these are assigned to program points. Since the analysis is to be used in some automatic tool, for example a compiler, it is important that it should be efficiently implemented. The main source of inefficiency arises in the analysis of looping constructs where the analysis has to iterate. As we have already seen, correctness of a program analysis amounts to checking that the results of the analysis give a safe approximation of the program's behaviour.

In the remainder of this paper we briefly review three approaches to program analysis:

– Data Flow Analysis
– Abstract Interpretation
– Types and Effects

These three approaches are illustrated by the papers included in this special section. This paper concludes by summarising each of the contributions.

1.2 Data Flow Analysis

Data flow analysis is the best known of the approaches to program analysis that we consider. The parity analysis from Section 1.1 is an example. Data flow analysis is treated in depth in standard compilers text books [1]. Hecht [7] gives an early text book treatment of data flow analysis techniques.

The starting point for data flow analysis is some representation of the control flow graph of the program – the nodes of such a graph may represent individual statements, as in a flowchart, or sequences of statements. The data flow analysis is usually specified as a set of equations which associate analysis information with program points. Program points correspond to nodes in the graph. Analysis information may be propagated forwards through the program, as in the parity analysis, or backwards.

Implementation of the analysis involves finding a solution to the data flow equations. A program point might have a number of predecessors (or successors for backward analysis): the information from these different predecessors (successors) must be combined. The combination operator is usually a union operation or an intersection operation. In the former case, we collect all of the information from the predecessors (successors); in the latter case we collect information that is common to all of the predecessors (successors). When we use the union combination operator, we require the least solution to the set of equations; when we use the intersection combination operator, we require the greatest solution.

Example 1. Reaching Definitions determines which set of definitions (assignments) are current when control reaches a certain program point. The analysis can be specified by equations of the following form:

$$RD_{entry}(p) = \begin{cases} \iota & \text{if } p \text{ is initial} \\ \bigcup_{p' \in pred(p)} RD_{exit}(p') & \text{otherwise} \end{cases}$$
$$RD_{exit}(p) \;= (RD_{entry}(p) \backslash kill(p)) \;\cup\; gen(p)$$

where RD_{entry} represents analysis information at the entry to a program point; RD_{exit} represents analysis information at the exit from a program point; *pred* is a predecessor function which returns all program points which immediately precede the given program point (in some execution sequence); and ι is an initial value. Each program point "kills" some definitions (those which define the same variable as the program point) and "generates" new definitions. A suitable representation for properties is sets of pairs where each pair is a variable and a program point – the meaning of the pair (x, p) is that the assignment to x at point p reaches this point. The initial value in this case is:

$$\iota = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

Reaching Definitions is a forwards analysis and we require the least solution to the set of equations.

Reaching definitions is just one of a large number of analyses that have been developed for imperative programming languages. Other examples include:

Available Expressions: detects which previously evaluated expressions are still valid (i.e. no variables used in the expression have been re-assigned since the expression was evaluated).
Live Variables: detects which of the variables assigned at a point will be used before being re-assigned a value.
Busy Expressions: identifies expressions which will be evaluated on all control paths from a point.
Constant Propagation: detects which variables have a constant value no matter which control path is used to arrive at a point.

Many others can be found in standard texts on compiler design.

The implementer is apparently faced with writing an algorithm to implement each of these different analyses. However, the different analyses have many features in common. This has led researchers to the definition of the notion of a *data flow framework* which specific analyses are instances of. The advantages of having a framework are:

1. **Correctness:** one can prove generic properties of the framework.
2. **Implementation:** one can construct parameterised, efficient implementations.

Reaching definitions may be presented as an instance of a *monotone framework*. Monotone frameworks consist of:

- a complete lattice of data flow values which satisfies some additional properties; in the reaching definition case, the lattice is the powerset of pairs of variables and the labels of assignment statements in a program.
- a set of monotone functions including the identity and closed under composition; used to represent the *transfer* functions associated with a program point.

Transfer functions specify how the statement(s) associated with a program point transforms data flow values. The combination operator is taken to be the least upper bound operator.

Formally, an instance of a framework consists of:

1. the complete lattice
2. the space of functions
3. some representation of the control flow in the program (forwards or backwards)
4. a finite set of entry points (which may be the start of the program – for forward analyses – or the exit points – for backward analyses)
5. an initial value to be used at the entry points
6. a mapping assigning transfer functions to program points

Example 2. Reaching definitions for a program can be cast as an instance of a monotone framework in the following way:

lattice, L $\sqsubseteq$ $\bigsqcup$ $\perp$	$\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$ $\subseteq$ $\bigcup$ $\emptyset$
control flow	program graph
entry points	{start node}
initial value	$\{(x, ?) \mid x \text{ is a variable in the program}\}$
functions f_p	$\{f : L \to L \mid \exists l_k, l_g : f(l) = (l \backslash l_k) \cup l_g\}$ $f_p(l) = (l \backslash kill(p)) \cup gen(p)$

where **Var** are the variables in the program and **Lab** are the labels in the program; $\mathcal{P}$ is the powerset constructor.

The notion of monotone frameworks is due to Kam and Ullman [8]. This account is based on the presentation of monotone frameworks in [12] which is closely followed by [10] in this section. The other data flow analyses mentioned above can also be specified as instances of a monotone framework.

The specification of reaching definitions above is for an intra-procedural analysis. The situation becomes more complex if the language being analysed includes procedures. Accurate analyses then require some mechanism for tracking the calling context for a procedure; [15] is a seminal paper on interprocedural analysis.

There is an extensive literature on algorithms for solving data flow equations. A naive iterative algorithm, specialised for reaching definitions and only computing

```
INPUT:    A control flow graph

OUTPUT:   RD

METHOD:   Step 1:  Initialisation
                   for all program points, p do
                        RD(p) := ∅;
                   RD(1) := ι;

          Step 2:  Iteration
                   change := true;
                   while change do
                        change := false;
                        for all program points, p do
                           new := ⋃_{p' ∈ pred(p)} f(RD,p')
                           if RD(p) ≠ new then
                              change := true;
                              RD(p) := new;

USING:    f(RD,p) = (RD(p) \ kill(p)) ∪ gen(p);
```

Fig. 1. A naive iterative algorithm for reaching definitions

RD_{entry}[1], is shown in Figure 1. Some of the most efficient iterative algorithms use worklists to control the iteration [9,5]. An interesting alternative approach is based on the observation that data flow analyses can be specified as propositions in the modal mu-calculus; solving data flow analysis problems is then a problem of model checking [14,16].

The main emphasis of the literature on data flow analysis is on efficiency; scant attention has been paid to correctness issues.

1.3 Abstract Interpretation

Abstract interpretation provides a framework for specifying and reasoning about program analyses [4]. The "set" of properties that is used in a program analysis is formally related to a set of run-time values that programs operate over. One way of establishing this relationship is via a Galois connection: a pair of functions (α, γ), the first of which is an *abstraction* map and the second of which is a *concretisation* map. The pair is required to satisfy the following:

$$\begin{array}{l} \alpha \circ \gamma \sqsubseteq id \\ \gamma \circ \alpha \sqsupseteq id \end{array}$$

where $\sqsubseteq$ is an approximation ordering on functions and *id* is the identity.

An oft cited example is "rule of signs" arithmetic. The set of properties is $\{\bot, 0, +, -, ?\}$ representing undefined, zero, positive, negative and all integers respectively. The ordering on these elements is shown in Figure 2.

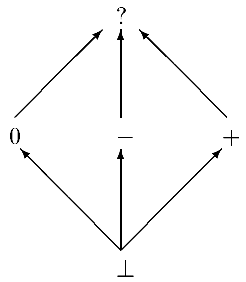

Fig. 2. The lattice of signs

The concretisation map is:

$$\begin{array}{l} \gamma : \{\bot, 0, +, -, ?\} \to \mathcal{P}(\mathcal{Z}) \\ \gamma(\bot) = \emptyset \\ \gamma(0) \ = \{0\} \\ \gamma(+) = \{z \mid z > 0\} \\ \gamma(-) = \{z \mid z < 0\} \\ \gamma(?) \ = \mathcal{Z} \end{array}$$

where $\mathcal{Z}$ is the integers and $\mathcal{P}$ is the powerset constructor. The abstraction map is defined as follows:

$$\begin{array}{l} \alpha : \mathcal{P}(\mathcal{Z}) \to \{\bot, 0, +, -, ?\} \\ \alpha(S) = \sqcap\{s \mid S \subseteq \gamma(s)\} \end{array}$$

Having formalised the relationship between properties and concrete values, the framework then prescribes how safe interpretations of operators can be constructed. In this context safety means that optimisations based on the analysis are conservative. For the rule of signs example, a safe interpretation for multiplication is $\otimes$:

$$\begin{array}{ll} 0 \otimes 0 \ = 0 & + \otimes + = + \\ - \otimes - = + & ? \otimes ? \ = ? \\ + \otimes - = - & + \otimes ? \ = ? \\ 0 \otimes + = 0 & 0 \otimes - = 0 \\ 0 \otimes ? \ = 0 & - \otimes ? \ = ? \\ \bot \otimes s \ = \bot & \end{array}$$

(where we have omitted the symmetric cases).

Program analysis based on abstract interpretation amounts to interpreting the program using the properties as data and the abstract operations.

The advantage of working within the framework is that correctness of any analysis follows almost by construction – the user is left to verify "local" conditions such as the safety of the abstract operations.

Since abstract interpretation involves running the program, program analysis can be seen as a form of fixed point iteration. This is similar to the situation that we have described for data flow analysis. Recall that the fixed point of a continuous function is the least upper bound of the sequence $(f^n(\bot))_n$. In the case of infinite sets of properties, and even in the finite case, iteration can be costly (in the infinite case, it may not even terminate). Cousot and Cousot [4] introduce the notion of

[1] The RD_{exit} information can be easily computed from this.

widening to accelerate convergence of such iterations. An operator $\nabla : L \times L \rightarrow L$ (where $(L, \sqsubseteq)$ is a complete lattice) is a widening operator if and only if:

- $\forall l_1, l_2[l_1 \sqsubseteq l_1 \nabla l_2 \sqsupseteq l_2]$ – it always returns an element larger than its two operands, and
- for all ascending chains $(l_n)_n$, the sequence $(s_n)_n$ defined by:
$$s_n = \begin{cases} l_n & \text{if } n = 0 \\ s_{n-1} \nabla l_n & \text{if } n > 0 \end{cases}$$
is an ascending chain which eventually stabilises.

Example 3. A classical example of widening is defined over the lattice of integer intervals:

$$L = \{\bot\} \cup \\ \{[z_1, z_2] \mid z_1 \leq z_2, z_1 \in \mathcal{Z} \cup \{-\infty\}, z_2 \in \mathcal{Z} \cup \{\infty\}\}$$

with

$$l_1 \sqsubseteq l_2 \;\Leftrightarrow\; \mathit{min}(l_2) \leq \mathit{min}(l_1) \;\wedge\; \mathit{max}(l_1) \leq \mathit{max}(l_2)$$

where

$$\mathit{min}(l) = \begin{cases} \infty \text{ if } l = \bot \\ z_1 \text{ if } l = [z_1, z_2] \end{cases}$$
$$\mathit{max}(l) = \begin{cases} -\infty \text{ if } l = \bot \\ z_2 \quad \text{if } l = [z_1, z_2] \end{cases}$$

Notice that L has infinite ascending chains. A widening operator for this lattice is [4]:

$$[z_1, z_2] \nabla [z_3, z_4] = [\mathit{low}, \mathit{high}]$$

where

$$\mathit{low} = \begin{cases} -\infty \text{ if } z_3 < z_1 \\ z_1 \quad \text{otherwise} \end{cases}$$

and

$$\mathit{high} = \begin{cases} \infty \text{ if } z_4 > z_2 \\ z_2 \text{ otherwise} \end{cases}$$

If we can define a suitable widening, the iterates in a fixed point computation can be combined using the widening rather than the least upper bound operator. Iteration involving a widening is guaranteed to converge; the trade-off is that the analysis may converge to an answer that is only a (safe) approximation of the true answer. Cousot and Cousot also introduce a notion of *narrowing* which allows refinement of such approximations to produce more accurate approximations. Widenings and narrowings can also be used in the iterative algorithms used for data flow analysis. The PAG system [10] allows the user to specify widenings and narrowings.

1.4 Types and Effects

Types have a long history in Computer Science; they appeared implicitly in FORTRAN (via a variable naming convention) and then explicitly in the ALGOL languages. A type system specifies how types are assigned to parts of a program. Type systems effectively specify the static semantics (or aspects of context-sensitive syntax) of the language – preventing integers being called with parameters, characters being added to reals, etc.. To be more concrete, consider the following fragment of a functional language:

$$x \in \mathit{Variable}$$
$$e \in \mathit{Expression}$$

$$\begin{aligned} e ::= {} & x \mid \\ & e\ e \mid \\ & \mathit{fun}\ x.e \end{aligned}$$

Terms in this language are (bound) variables, application terms or terms represention functions (abstractions) of one parameter.

A suitable syntax for types for this small language might be:

$$\alpha \in \mathit{Type\text{-}variable}$$
$$\tau \in \mathit{Type}$$

$$\tau ::= \alpha \mid \tau \rightarrow \tau$$

where α is a type variable and types of the form $\tau \rightarrow \tau$ represent function types. Thus a suitable type for the identity function ($\mathit{fun}\ x.x$) is $\alpha \rightarrow \alpha$, whilst a suitable type for:

$$\mathit{fun}\ f.(\mathit{fun}\ x.f\ x)$$

(the "apply" function) is:

$$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

Notice that these types are not unique; other types for the identity include:

$$(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$
$$((\alpha \rightarrow \beta) \rightarrow \gamma) \rightarrow ((\alpha \rightarrow \beta) \rightarrow \gamma)$$
$$\vdots$$

but in some sense, the first type that we gave was the minimal type.

Type systems are often specified by sets of axioms and inference rules. The type system for our small language is specified, relative to some environment (giving types to free variables), by the set of axioms and rules shown in Figure 3; A_x represents the environment A with the entries for x removed.

The axiom says that the environment must be accessed in order to find the type of a variable. The first rule says that the type of an application term is τ_2 if the function sub-term has result type τ_2 and the type of the argument sub-term matches the argument type of the function. The final rule assignes a function type to an abstraction: if the body of the function can be assigned type τ_2 under the assumption that the argument has type τ_1 then the abstraction has type $\tau_1 \rightarrow \tau_2$.

The language can be equipped with a reduction semantics. Correctness of the type system may then be expressed as a subject reduction property: if e is assigned

$$A \vdash x : \tau \qquad \text{if } A(x) = \tau$$

$$\frac{A \vdash e_1 : \tau_1 \to \tau_2 \qquad A \vdash e_2 : \tau_1}{A \vdash e_1\ e_2 : \tau_2}$$

$$\frac{A \cup \{x : \tau_1\} \vdash e : \tau_2}{A_x \vdash \mathit{fun}\ x.e : \tau_1 \to \tau_2}$$

Fig. 3. A type system

$$
\begin{array}{lll}
\mathcal{W}(A, x) & = (A(x), \mathit{id}) & \\
\mathcal{W}(A, e_1\ e_2) & = (S_3(\alpha), S_3 \circ S_2 \circ S_1) & \\
\text{where} & & \\
(\tau_1, S_1) & = \mathcal{W}(A, e_1) & \\
(\tau_2, S_2) & = \mathcal{W}(S_1\ A, e_2) & \\
S_3 & = \mathcal{U}(\tau_2 \to \alpha, S_2(\tau_1)) & \alpha \text{ new} \\
\mathcal{W}(A, \mathit{fun}\ x.e) & = (S(\alpha) \to \beta, S) & \\
\text{where} & & \\
(\beta, S) & = \mathcal{W}(A \cup \{x : \alpha\}, e) & \alpha \text{ new}
\end{array}
$$

Fig. 4. A type inference algorithm

type τ and e reduces to e', then e' is also assigned type τ.

Refinements to the language and type system includes adding constants (both to expressions and types), parametric polymorphism and subtypes.

Milner [11] proposed an algorithm for type inference in the presence of polymorphism; it forms the basis for many of the algorithms used in type-based approaches to program analysis. The algorithm uses unification. A *unifier* of two types is a substitution on type variables which, when applied to the two types, makes them equal. The unification algorithm, $\mathcal{U}$, finds a unifier for τ_1 and τ_2 or fails. Furthermore

1. If $\mathcal{U}(\tau_1, \tau_2) = V$ then $V\ \tau_1 = V\ \tau_2$, i.e. V unifies τ_1 and τ_2.
2. If S unifies τ_1 and τ_2 then $\mathcal{U}(\tau_1, \tau_2)$ returns some V and there is another substitution R such that:

   $$S = R\ V$$

   which amounts to stating that V does the least amount of work to equate the two types; V is called the *most general unifier*.
3. V only involves variables in τ_1 and τ_2; no new variables are introduced during unification.

Given an expression and an environment, Milner's algorithm produces a type and a substitution (on type variables). The algorithm, as applied to our language fragment, is shown in Figure 4.

Example 4.

$$\mathcal{W}(\emptyset, \mathit{fun}\ x.x) = (\alpha \to \alpha, \mathit{id})$$

since

$$\mathcal{W}(\{x : \alpha\}, x) = (\alpha, \mathit{id})$$

Example 5. A more complex example is:

$$\mathcal{W}(\emptyset, \mathit{fun}\ f.\mathit{fun}\ x.f\ x) = ((\beta \to \gamma) \to \beta \to \gamma, [\alpha := \beta \to \gamma])$$

since

$$\mathcal{W}(\{f : \alpha\}, \mathit{fun}\ x.f\ x) = (\beta \to \gamma, [\alpha := \beta \to \gamma])$$

and

$$\mathcal{W}(\{f : \alpha, x : \beta\}, f\ x) = (\gamma, [\alpha := \beta \to \gamma])$$

because

$$
\begin{array}{ll}
\mathcal{W}(\{f : \alpha, x : \beta\}, f) & = (\alpha, \mathit{id}) \\
\mathcal{W}(\{f : \alpha, x : \beta\}, x) & = (\beta, \mathit{id}) \\
\mathcal{U}(\beta \to \gamma, \alpha) & = [\alpha := \beta \to \gamma]
\end{array}
$$

Increasingly, type systems have been used in program analysis. In this work, "non-standard" types are used. For example, [6] applies these ideas to strictness analysis for lazy functional languages. Strictness is an *extensional* property: it concerns the input/output behaviour of a function – if the input is undefined, the output is. Type systems are well-suited to describe such properties. In the context of program analysis, we are generally more interested in *intensional* properties: *how* the function achieves its result rather than *what* the result is. In order to infer intensional information, type constructors may be annotated with effects and effects may also be inferred in conjunction with the type information. Such enhanced type systems are sometimes referred to as Annotated Type and Effect Systems. The paper on behaviours [2] elaborates on the details of one such system.

2 About this issue

2.1 CML Behaviours

Amtoft, Nielson and Nielson [2] present a system for deriving behaviours from CML programs. CML is a functional language, based on SML, extended with concurrency primitives for creating and communicating across channels and spawning new processes. Behaviours are terms in a process algebra which describe the communication behaviour of the program. Behaviours may then be analysed to determine certain properties of the program under analysis. One such property is *communication topology*: for example, analysis of the behaviour can determine whether the topology is finite.

The tool is based on a type-based approach. The paper specifies an annotated types and effects system for inferring behaviours. Judgements are of the form:

$$C, A \vdash e : t\ \&\ b$$

where C is a set of constraints, A is an environment, e is the program (expression), t is a type and b is the behaviour. Type inference is implemented using a modification of the $\mathcal{W}$ algorithm of [11].

The authors illustrate the use of their tool by applying it to a standard benchmark problem.

2.2 Program Analysis for Logic Programming Languages

Codish, Demoen and Sagonas [3] present a tool for the analysis of logic programs. The basic mechanism involves tabulating the calls to and corresponding answers from certain predicates. The authors develop their approach in a number of steps:

1. **Abstract Compilation and Concrete Tabulation:** in which the subject program is first "compiled" into an abstract form and then interpreted (with the results of interpretation being tabulated). This corresponds to a naive abstract interpretation.
2. **Abstract Compilation and Abstract Tabulation:** in which operations used in the interpretation (such as unification) are also abstracted.
3. **Alternative Collecting Strategies:** in which, rather than using abstract domains based on powersets (as in the previous two steps), other domains are used. The domains may be infinite, in which case there is a need for widening operators. At the very least the analysis must specify an abstract least upper bound operation.
4. **Beyond Abstract Compilation:** in which the authors discuss generalisations of the substitution mechanism used in the previous steps; one particular generalisation is to allow the modelling of relationships bewteen variables.

The different steps are well-illustrated using standard analyses that have appeared in the literature. The analyses are implemented in the logic programming system XSB [13].

2.3 A Program Analysis Generator

Martin [10] introduces the PAG system. PAG is a program analysis generator that generates interprocedural data flow analysers. The viewpoint of the author is that data flow analysis is a special case of abstract interpretation; this is quite clearly reflected in the features of the tool. PAG provides two specification languages: DATLA and FULA.

DATLA is the language used for defining the abstract domains of an analysis. It is a functional language. It is assumed that abstract domains are complete lattices. DATLA provides a number of basic domains and operations for combining domains.

FULA is the language for describing transfer functions. It is a first-order, monomorphic functional language with eager semantics.

PAG assumes that the program is represented as a control flow graph. In addition to the standard parameters for a monotone data flow framework (see earlier), the user is able to specify the fixed point algorithm to be used (from a fixed menu of worklist-based algorithms) and widening and narrowing operations.

References

1. A. Aho, R. Sethi and J. Ullman, *Principles of Compiler Design*, Addison Wesley, 1986.
2. T. Amtoft, H. R. Nielson and F. Nielson, *Behaviour Analysis for Validating Communication Patterns*, STTT (this issue).
3. M. Codish, B. Demoen and K. Sagonas, *Semantics-based Program Analysis for Logic-based Languages using XSB*, STTT (this issue).
4. P. Cousot and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs*, in *Proc. POPL'77*, ACM Press, 1977.
5. C. Fecht and H. Seidl, *An even faster solver for general systems of equations*, in *Proc. SAS'96*, Springer Verlag, LNCS 1145, 1996.
6. C. Hankin and D. Le Métayer, *Deriving Algorithms from Type Inference Systems*, in *Proc. POPL'94*, ACM Press, 1994.
7. M. S. Hecht, *Flow analysis of computer programs*, North Holland, 1977.
8. J. B. Kam and J. D. Ullman, *Monotone data flow analysis frameworks*, Acta Informatica, 7, 305–317, 1977.
9. G. Kildall, *A Unified Approach to Global Program Optimization*, in *POPL'73*, ACM Press, 1973.
10. F. Martin, *PAG – An Efficient Program Analyzer Generator*, STTT (this issue).
11. R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, 17, 348–375, 1978.
12. F. Nielson, H.R. Nielson and C. Hankin, *Principles of Program Analysis - Flows and Effects*, to appear.
13. K. Sagonas, T. Swift and D. S. Warren, *XSB as an Efficient Deductive Database Engine*, in *Proc ACM SIGMOD Conf on the Management of Data*, ACM Press, 1994.
14. D. Schmidt, *Data Flow Analysis is Model Checking of Abstract Interpretations*, in *Proc. POPL'98*, ACM Press, 1998.
15. M. Sharir and A.Pnueli, *Two approaches to interprocedural data flow analysis*, in S. S. Muchnick and N. D. Jones (editors), *Program Flow Analysis*, Prentice Hall International, 1981.
16. B. Steffen, *Generating Data Flow Analyses from Modal Specifications*, Science of Computer Programming, 21, 1993.