Flow Logics for Constraint Based Analysis

Hanne Riis Nielson and Flemming Nielson

Department of Computer Science, Aarhus University, Denmark
e-mail: hrn@daimi.aau.dk fn@daimi.aau.dk

Abstract. Flow logic offers a compact and versatile notation for expressing the acceptability of solutions to program analysis problems. In contrast to previous logical formulations of program analysis it aims at integrating existing approaches to data flow analysis and control flow analysis. It is able to deal with a broad variety of language paradigms, program properties, kinds of formal semantics, and methods used for computing the best solution.

In this paper we illustrate how a compositional flow logic (in "succinct" form) can be systematically transformed into an efficient exhaustive procedure for computing the best solution of a set of constraints generated. This involves transformations to attribute grammars and to specifications of the ("verbose") form used in control flow analysis.

Keywords. Program analysis, data flow analysis, control flow analysis, constraint based analysis, attribute grammars.

1 Introduction

Background. The development of program analyses [18] for complex languages with procedures, assignments, pointers, block structure and communication is no easy task. One facet of this is that the resulting analyses are often too unwieldy for human consumption and for formal verification with respect to the semantics of the language; this increases the likelihood that the analyses are not completely safe and that the resulting systems have security loop-holes. Another facet is that it is hard to implement the analyses so as to yield acceptable time and space performance; this reduces the usefulness of the analyses and may lead to the use of cheaper analyses that are overly approximate.

Research aimed at overcoming the "unwieldiness" problem often suggests the use of compositional or syntax-directed specifications. One popular approach is that of type systems perhaps extended with annotations concerning side effects [23] or with properties of the states consumed and produced. Another popular approach for functional and object-oriented languages is the generation of constraints for expressing e.g. the connection between the states consumed and produced [8]. As we shall see in this paper this is related to the use of circular attribute grammars [1, 2] for specifying program analyses.

Research aimed at overcoming the "efficiency" problem often studies ways of presenting the flow equations in a manner independent of the syntax of the programming language. This may take the form of traditional data flow equations [7] that can then be interfaced with a solver that traverses the equations in a specific order (e.g. reverse postorder) so as to yield good time and space performance.

The two research directions need to cooperate although the different nature of the problems studied makes it hard to succeed: while compositionality usually is a good strategy for overcoming the "unwieldiness" (because it allows us to grasp one thing at a time) it is seldom a good strategy for achieving the "efficiency" (because attributes that influence one another are not necessarily close in terms of the syntax and hence the propagation of the attributes becomes unnecessarily costly). Indeed a lot of the research on overcoming the penalties of using type systems can be understood as identifying techniques for abandoning compositionality [22, 5].

Flow logics. The notion of flow logic in *succinct* form was devised in [17] for facilitating the integration of interprocedural data flow analysis techniques with those of control flow analysis, taking care of side-effects, multiple environments, and multiple analysis contexts while still obtaining a syntax-directed specification. As such the notation is directly aimed at overcoming the "unwieldiness" problem in allowing to combine program analyses that traditionally are presented using different techniques. Hence flow logic distinguishes itself from other logical based approaches to program analysis (e.g. [12, 3, 10]) in that it aims at combining *existing* approaches to program analyses rather than offering yet another approach.

Flow logics (in succinct or verbose form) have been specified for a variety of programming language paradigms; in addition to the functional and imperative paradigms mentioned above it has been used for languages and calculi supporting concurrency and objects.

Contribution. In this paper we first present an example flow logic in succinct form. We then show that minor transformations (little more than a redefinition of LaTeX macros) allow us to obtain a specification in the form of an extended attribute grammar [26]. Other transformations (mainly for making program points explicit) suffice for obtaining a specification in the form of a circular attribute grammar or a control flow analysis (a set based analysis). However, all of these specifications are to be viewed as recipes for *verifying* whether or not a proposed solution to the program analysis problem is indeed acceptable.

To *compute* the best (in our formulation this means the smallest) solution we shall base ourselves on one of the more *verbose* formulations obtained above but we shall abstain from directly using it as a method for computing the best solution. Instead we show how to obtain a finite set of conditional constraints; this is possible due to the compositional nature of the specification (unlike those of e.g. [16]). The finite set of constraints can then be solved using graph based techniques and possibly using specific iteration orders [4]. We shall see that this leads to the best solution according to the specification.

The development is illustrated on a simplified fragment of the specification developed in [17]. The full specification deals with a functional language with side-effects (in the style of Standard ML [15]) and studies different ways of establishing context (including call strings [20] and data dependencies [13, 19]). The fragment considered here just deals with a simplified untyped functional language and only studies contexts in the form of call strings.

2 Succinct Flow Logics

To illustrate the development we shall make use of the syntax

$$e ::= x \mid \texttt{fn}_\pi\ x \texttt{ => } e_0 \mid (e_1\ e_2)^l \mid \cdots$$

where $x \in \mathsf{Var}$, $\pi \in \mathsf{Pnt_F}$ and $l \in \mathsf{Lab}$. The nature of the unspecified expressions will be of little concern to us except that they allow to assign to variables and to access their values; this means that the analysis needs to be able to deal with side-effects. Since the language is untyped we can use function definition and application to encode the fixed point combinator and hence express recursive functions as well. The purpose of the domains Lab (of labels) and $\mathsf{Pnt_F}$ (of function definition points) will be explained below.

Example 1. As a running example we shall consider the program

$$((\texttt{fn}_x\ \texttt{x => ((x x)}^1\ \texttt{(fn}_y\ \texttt{y => y))}^2\texttt{)}\ \texttt{(fn}_z\ \texttt{z => z))}^3$$

Execution of this program proceeds as follows: At the application point 3, `x` is bound to $\texttt{fn}_z$ `z => z` and we get $(((\texttt{fn}_z\ \texttt{z => z})\ (\texttt{fn}_z\ \texttt{z => z}))^1\ (\texttt{fn}_y\ \texttt{y => y}))^2$. At the point 1, `z` is bound to $\texttt{fn}_z$ `z => z` so the next step of reduction gives $((\texttt{fn}_z\ \texttt{z => z})\ (\texttt{fn}_y\ \texttt{y => y}))^2$. Finally, at the point 2, `z` is bound to $\texttt{fn}_y$ `y => y` and we obtain the result $\texttt{fn}_y$ `y => y`. □

Abstract domains. The analysis is a combined closure and reference cell analysis: for each subexpression it is determined which closures and cells it may evaluate to; one aspect of this involves tracking those abstract values that variables and cells can evaluate to. To present the analysis we shall use the abstract domains defined in Table 1. The domain Mem (of mementa or contexts) facilitates representing call strings of length at most k; here a call string [20] is a list of labels $l_1, \cdots, l_n$ denoting that the last n calls were of the form $(e_{1n}\ e_{2n})^{l_n}, \cdots, (e_{11}\ e_{21})^{l_1}$ with $(e_{11}\ e_{21})^{l_1}$ being the most recent call; we write ϵ for the empty call string.

There are three ways of constructing the basic abstract values (in $\mathsf{Val_B}$). For the development to be performed here it is not important how data like integers and booleans are analysed and hence we have chosen a trivial domain Data for this.

$$\begin{array}{rl} m \in \mathsf{Mem} & = \mathsf{Lab}^{\leq k} \\ d \in \mathsf{Data} & = \{\diamond\} \\ (\pi, m_d) \in \mathsf{Closure} & = \mathsf{Pnt_F} \times \mathsf{Mem} \\ (\varpi, m_d) \in \mathsf{Cell} & = \mathsf{Pnt_R} \times \mathsf{Mem} \\ v \in \mathsf{Val_B} & = \mathsf{Data} \cup \mathsf{Closure} \cup \mathsf{Cell} \\ W \in \widehat{\mathsf{Val}} & = \mathcal{P}(\mathsf{Mem} \times \mathsf{Val_B}) \\ R \in \widehat{\mathsf{Env}} & = \mathsf{Var} \to \widehat{\mathsf{Val}} \\ S \in \widehat{\mathsf{Store}} & = \mathsf{Cell} \to (\widehat{\mathsf{Val}} \times \{\mathsf{O}, \mathsf{I}, \mathsf{M}\}) \end{array}$$

Table 1. Abstract domains.

Functions (or closures) are represented by pairs consisting of the function point π where the function was constructed and the context m_d in which this took place; as we shall see we will have other means for recording the abstract values of the free variables in the function. In a similar way assignable values (or cells) are represented by pairs consisting of the reference point $\varpi \in \mathsf{Pnt_R}$ where the cell was created (e.g. by a ML-like $\mathtt{ref}_\varpi$ construct) and the context m_d in which this took place.

Example 2. Let us consider the case $k = 1$, i.e. call strings have length at most 1. The three function definitions of the example program of Example 1 can be represented by (x, ϵ), $(y, 3)$ and (z, ϵ). This reflects that the functions x and z are defined before the first application whereas the function y only is defined after the application at point 3 has been performed. □

A subexpression may evaluate to several (abstract) values in different contexts. To record this we use the domain $\widehat{\mathsf{Val}}$ that is isomorphic to $\mathsf{Mem} \to \mathcal{P}(\mathsf{Val_B})$ and for each context m_d the value $W \in \widehat{\mathsf{Val}}$ gives the set $W(m_d) \subseteq \mathsf{Val_B}$ of basic values that the subexpression may evaluate to. In a similar way the domain $\widehat{\mathsf{Env}}$ of (abstract) environments associates a value to each variable.

Example 3. As shown in Example 1, the variable z is bound to different values during the execution. In the analysis its intended abstract value can be written $\{(1, (z, \epsilon)), (2, (y, 3))\}$ reflecting that at the application point 1, z is bound to the function z (defined in the context ϵ) and at the point 2 it is bound to the function y (defined in the context 3). □

The main point of the domain $\widehat{\mathsf{Store}}$ of (abstract) stores is to associate a value to each cell. Additionally it records how many references (O for zero, I for zero or one, and M for any number) there is to a given cell; this facilitates the full analysis [17] to model destructive updating in the manner of [25]. The domains $\widehat{\mathsf{Val}}$, $\widehat{\mathsf{Env}}$, and $\widehat{\mathsf{Store}}$ are partially ordered in the obvious way (with $\mathsf{O} \sqsubseteq \mathsf{I} \sqsubseteq \mathsf{M}$) and this turns them into complete lattices.

$$\begin{aligned}
\mathcal{W}_F &\in \widehat{\mathsf{WCache}}_\mathsf{F} = (\bullet\mathsf{Pnt}_\mathsf{F} \cup \mathsf{Pnt}_\mathsf{F}\bullet) \rightarrow \widehat{\mathsf{Val}} \\
\mathcal{S}_F &\in \widehat{\mathsf{SCache}}_\mathsf{F} = (\bullet\mathsf{Pnt}_\mathsf{F} \cup \mathsf{Pnt}_\mathsf{F}\bullet) \rightarrow \widehat{\mathsf{Store}} \\
\mathcal{R}^\mathsf{d}_F, \mathcal{R}^\mathsf{c}_F &\in \widehat{\mathsf{RCache}}_\mathsf{F} = \mathsf{Pnt}_\mathsf{F} \rightarrow \widehat{\mathsf{Env}} \\
\mathcal{M}_F &\in \widehat{\mathsf{MCache}}_\mathsf{F} = \mathsf{Pnt}_\mathsf{F} \rightarrow \mathcal{P}(\mathsf{Mem})
\end{aligned}$$

Table 2. Caches.

Caches. Since the analysis will be specified in a compositional (syntax-directed) manner we need additional machinery ("global attributes") for transferring information from one part of the program to another. To do so we shall make use of the caches defined in Table 2:

- The *value cache* $\mathcal{W}_F$ has two components: $\mathcal{W}_F(\bullet\pi)$ records the actual parameters to the function labelled π (and is used to link the actual parameter to the formal parameter); $\mathcal{W}_F(\pi\bullet)$ records the results of evaluating the function body (and is used to link the result back to the call site).
- The *store cache* $\mathcal{S}_F$ is in many ways similar: $\mathcal{S}_F(\bullet\pi)$ records the stores possible at the function call; $\mathcal{S}_F(\pi\bullet)$ records the stores possible after the evaluation of the function body.
- The *environment caches* $\mathcal{R}^\mathsf{d}_F$ and $\mathcal{R}^\mathsf{c}_F$ are needed because we allow nested functions and insist on static scope also for the free variables of functions: $\mathcal{R}^\mathsf{d}_F(\pi)$ records the environment in force when the function labelled π is declared; $\mathcal{R}^\mathsf{c}_F(\pi)$ records the same environment but modified to the context in which the function body is evaluated.
- The *memento cache* $\mathcal{M}_F$ is used to ensure that a function is in fact analysed only once (for all contexts of relevance) rather than many times (one context at a time): $\mathcal{M}_F(\pi)$ records the set of contexts in which the function body needs to be analysed.

Note that if we simplify the analysis to ignore context (à la 0-CFA analyses) then the memento cache would merely record whether or not the function body is reachable (as in [6]).

Example 4. For the program of Example 1 we may take the following caches:

π	x	y	z
$\mathcal{W}_F(\bullet\pi)$	$\{(3,(z,\epsilon))\}$	$\emptyset$	$\{(1,(z,\epsilon)),(2,(y,3))\}$
$\mathcal{W}_F(\pi\bullet)$	$\{(3,(y,3))\}$	$\emptyset$	$\{(1,(z,\epsilon)),(2,(y,3))\}$
$\mathcal{S}_F(\bullet\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{S}_F(\pi\bullet)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{R}^\mathsf{d}_F(\pi)$	$[\,]$	$[\mathtt{x} \mapsto \{(3,(z,\epsilon))\}]$	$[\,]$
$\mathcal{R}^\mathsf{c}_F(\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{M}_F(\pi)$	$\{3\}$	$\emptyset$	$\{1,2\}$

(It turns out that this is indeed the best analysis of the example program.) □

$$R, M \;\rhd\; x : S \to S \;\&\; R(x)$$

$$\begin{array}{l} R, M \;\rhd\; \mathbf{fn}_\pi\; x \texttt{=>} e_0 : S \to S \;\&\; \{(m, (\pi, m)) \mid m \in M\} \\ \mathsf{iff}\;\; \mathcal{R}^{\mathsf{c}}_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \;\rhd\; e_0 : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \;\&\; \mathcal{W}_F(\pi\bullet) \;\wedge \\ \quad R \sqsubseteq \mathcal{R}^{\mathsf{d}}_F(\pi) \end{array}$$

$$\begin{array}{l} R, M \;\rhd\; (e_1\; e_2)^l : S_1 \to S_4 \;\&\; W \\ \mathsf{iff}\;\; R, M \;\rhd\; e_1 : S_1 \to S_2 \;\&\; W_1 \;\;\wedge\;\; R, M \;\rhd\; e_2 : S_2 \to S_3 \;\&\; W_2 \;\;\wedge \\ \quad \forall \pi \in \{\pi \mid (m, (\pi, m_d)) \in W_1\} : \\ \qquad W_2\lceil X^l_{\mathsf{hc}}(M)\rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge S_3\lceil X^l_{\mathsf{hc}}(M)\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \;\wedge \\ \qquad \mathcal{W}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(M)\rceil \subseteq W \wedge \mathcal{S}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(M)\rceil \sqsubseteq S_4 \;\wedge \\ \qquad \mathcal{R}^{\mathsf{d}}_F(\pi)\lceil X^l_{\mathsf{dc}}(M, W_1, \pi)\rceil \sqsubseteq \mathcal{R}^{\mathsf{c}}_F(\pi) \wedge \mathsf{take}_k(l, M) \subseteq \mathcal{M}_F(\pi) \\ \quad \text{for some } S_2, S_3, W_1, W_2 \end{array}$$

Table 3. Succinct flow logic for the functional fragment.

Specification of the analysis. In the current specification of the analysis we are not concerned with computing the best solution (see later) but merely with verifying whether or not a proposed solution is acceptable in the sense that no errors will occur when performing transformations based on it. We express this as follows:

$$(\mathcal{R}^{\mathsf{d}}_F, \mathcal{R}^{\mathsf{c}}_F, \mathcal{M}_F, \mathcal{S}_F, \mathcal{W}_F) \text{ satisfies } R, M \;\rhd\; e : S_1 \to S_2 \;\&\; W$$

Here the proposed solution consists of the five caches of Table 2 and the entities R, M, S_1, S_2 and W: $R \in \widehat{\mathsf{Env}}$ is the environment in which e is to be analysed, $M \in \mathcal{P}(\mathsf{Mem})$ is the set of contexts in which e is to be analysed, $S_1 \in \widehat{\mathsf{Store}}$ is the store that is possible immediately before e, $S_2 \in \widehat{\mathsf{Store}}$ is the store that is possible immediately after e, and $W \in \widehat{\mathsf{Val}}$ is the value that e can evaluate to. Since the five caches of Table 2 remain "constant" throughout the verification we shall dispense with listing them when defining the "$\rhd$" relation in Table 3. Note that the clauses are defined compositionally and hence clearly are well-defined. We shall motivate the individual clauses below.

Example 5. Given the caches of Example 4 we may verify the following formula for the program of Example 1

$$[\,], \{\epsilon\} \rhd \mathsf{program} : [\,] \to [\,] \;\&\; \{(\epsilon, (y, 3))\}$$

reflecting that the initial environment is empty, that the initial context is the empty call string, that the program does not manipulate the store (which hence is empty) and that the final value is described by $\{(\epsilon, (y, 3))\}$. The verification will amount to a proof using the clauses of Table 3 as rules and axioms; if successful, the proof and the caches constitute the analysis of the program. □

The clause for *variables* merely demands that the store after x equals the store possible before x and that the value associated with x in the environment equals the value that x evaluates to.

The clause for *function definition* starts out in a similar way except that the value of the definition must equal

$$\{(m,(\pi,m)) \mid m \in M\}$$

because when the function definition is analysed for $m \in M$ the value produced will be (π, m). Next the environment relevant for the free variables is recorded for later usage (by means of $R \sqsubseteq \mathcal{R}^{\mathsf{d}}_F(\pi)$). Finally, the function body itself is analysed and this involves the information in four of the five caches of Table 2.

Example 6. As part of verifying the formula of Example 5 we will need to verify

$$[\,], \{\epsilon\} \rhd \mathtt{fn}_z \ \mathtt{z} \Rightarrow \mathtt{z} : [\,] \to [\,] \ \& \ \{(\epsilon,(z,\epsilon))\}$$

This follows from the clause for function definition because $[\,] \sqsubseteq [\,]$ and

$$[\mathtt{z} \mapsto \{(1,(z,\epsilon)),(2,(y,3))\}], \{1,2\} \rhd \mathtt{z} : [\,] \to [\,] \ \& \ \{(1,(z,\epsilon)),(2,(y,3))\}$$

follows from the clause for variables. Note that although the function z is called twice it is only "analysed" once. □

The clause for *function application* first performs the recursive calls for verifying the proposed solution with respect to the operator and operand. For each function π that could possibly be called, a number of conditions are verified. First that the value (W_2) of the argument is contained among the arguments ($\mathcal{W}_F(\bullet\pi)$) that π is called with, and similarly, that the store (S_3) that holds after evaluation of the argument is contained in the store ($\mathcal{S}_F(\bullet\pi)$) holding before the body of π. However, we do not simply write $W_2 \subseteq \mathcal{W}_F(\bullet\pi)$ and $S_3 \sqsubseteq \mathcal{S}_F(\bullet\pi)$ because the context changes between the call site and the function body. To take care of this we write

$$\begin{aligned} W\lceil Y \rceil &= \{(m_2,v) \mid (m_1,v) \in W, (m_1,m_2) \in Y\} \\ (S\lceil Y \rceil)(\varpi,m) &= (S(\varpi,m))\lceil Y \rceil \end{aligned}$$

for recording that contexts are to be changed from the corresponding first components of Y to the corresponding second components. In the case of transferring W_2 to $\mathcal{W}_F(\bullet\pi)$ the appropriate context change is expressed by

$$X^l_{\mathsf{hc}}(M) = \{(m, \mathsf{take}_k(l\hat{\ }m)) \mid m \in M\}$$

that simply prepends the label l of the call to all contexts and then truncates the length to at most k. The same change has to be performed for the store.

Continuing with the clause for function application it is verified that the value resulting from the function body ($\mathcal{W}_F(\pi\bullet)$) is contained in the overall value of the call (W) and that the store after the function body ($\mathcal{S}_F(\pi\bullet)$) is contained in the store after the call (S_4); for this the required change of context is expressed by

$$\begin{aligned} X^l_{\mathsf{ch}}(M) = {} & \{(m, \mathsf{drop}_1(m)) \mid \mathsf{drop}_1(m) \in M, \mathsf{take}_1(m) = l\} \\ & \cup \{(m, \mathsf{drop}_1(m)\hat{\ }l') \mid \mathsf{drop}_1(m)\hat{\ }l' \in M, \mathsf{take}_1(m) = l\} \end{aligned}$$

where $\mathsf{drop}_1(m)$ removes the first element of m, i.e. the label of the most recent call site. Next we verify that the function called will indeed be analysed in all relevant contexts (as given by $\mathsf{take}_k(l, M) = \{\mathsf{take}_k(l\hat{\ }m) \mid m \in M\}$). Finally we enforce static scope for the free variables in the function being called by ensuring that the definition time environment $(\mathcal{R}_F^{\mathsf{d}}(\pi))$ is contained in that of the function body $(\mathcal{R}_F^{\mathsf{c}}(\pi))$; for this the required change of context is

$$X^l_{\mathsf{dc}}(M, W, \pi) = \{(m_d, \mathsf{take}_k(l\hat{\ }m)) \mid m \in M, (m, (\pi, m_d)) \in W\}$$

Example 7. Verifying the formula of Example 5 also involves verifying

$$[\mathtt{x} \mapsto \{(3,(z,\epsilon))\}], \{3\} \rhd (\mathtt{x}\ \mathtt{x})^1 : [\,] \to [\,]\ \&\ \{(3,(z,\epsilon))\}$$

For this, the clause for application demands that we verify

$$[\mathtt{x} \mapsto \{(3,(z,\epsilon))\}], \{3\} \rhd \mathtt{x} : [\,] \to [\,]\ \&\ \{(3,(z,\epsilon))\}$$

which follows directly from the clause for variables. Only the function z can be called so we have to verify a number of conditions for this function, including that $\{(3,(z,\epsilon))\}\lceil X^1_{\mathsf{hc}}(\{3\})\rceil \subseteq \mathcal{W}_F(\bullet z)$ and $\mathcal{W}_F(z\bullet)\lceil X^1_{\mathsf{ch}}(\{3\})\rceil \subseteq \{(3,(z,\epsilon))\}$. Here $X^1_{\mathsf{hc}}(\{3\}) = \{(3,1)\}$ and the effect of the transformation will be to remove all pairs that do not have 3 as the first component and to replace the first components of the remaining pairs with 1. Similarly, $X^1_{\mathsf{ch}}(\{3\}) = \{(1,3)\}$ so in this case the transformation will remove pairs that do not have 1 as the first component (i.e. pairs that do not correspond to the current call point) and replace the first components of the remaining pairs with 3. It is now easy to verify that the above two conditions are fulfilled for the caches of Example 4. □

Flow dependencies. The interplay between the clauses for function application and function definition is illustrated in Figure 1. Here each of the caches $\mathcal{R}_F^{\mathsf{d}}$, $\mathcal{R}_F^{\mathsf{c}}$ and $\mathcal{M}_F$ are represented by rectangles and similarly the two "components" of the caches $\mathcal{W}_F$ and $\mathcal{S}_F$ are represented by rectangles. A judgement of the form $R, M\ \rhd\ e : S_1 \to S_2\ \&\ W$ is represented by a node

$$\circ\ \circ\ \circ\ e\ \ \circ\ \circ$$

where the three circles before e represent R, M and S_1 (in that order) and the two circles after e represent S_2 and W (in that order). An arrow indicates flow of information.

Containments versus equalities. Since the specification in Table 3 is concerned with verifying whether or not a proposed solution is acceptable it is sensible that the clause for function application employs a containment like $\mathsf{take}_k(l, M) \subseteq \mathcal{M}_F(\pi)$ rather than an equality like $\mathsf{take}_k(l, M) = \mathcal{M}_F(\pi)$. The reason is that there might be other instances of the clause where the label of the application is different but yet the same function is called. If $l_1, \cdots, l_n$ are all the labels

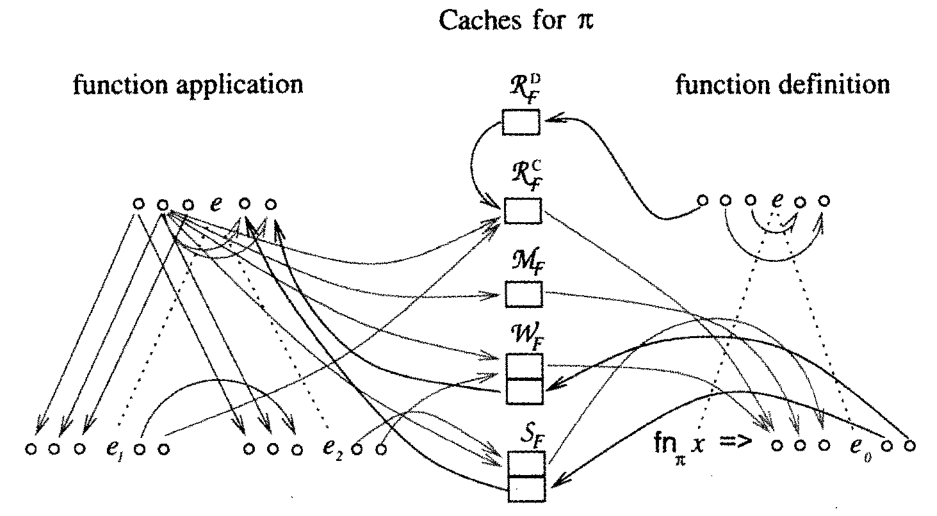

Fig. 1. The use of the caches.

of applications where a function labelled π is called then taking the smallest (i.e. least) solution ensures that $\mathcal{M}_F(\pi) = \mathsf{take}_k(l_1, M) \cup \cdots \cup \mathsf{take}_k(l_n, M)$ which is the desired result; as in [16, 6] one can prove that it is always possible to find an acceptable solution that is also the smallest one. In fact it would be incorrect to replace the containment by an equality: if $M \neq \emptyset$, $k > 0$ and $l_{i1} \neq l_{i2}$ then it is impossible to obtain $\mathsf{take}_k(l_i, M) = \mathcal{M}_F(\pi)$ for all i.

Although the clauses in Table 3 contain no explicit equalities they do contain a lot of implicit equalities because the same flow variable is used more than once in the same clause. One can avoid this by introducing new variables and then linking them explicitly by containments as illustrated below.

$R, M \ \rhd\ x : S_1 \to S_2 \ \&\ W$
iff $S_1 \sqsubseteq S_2 \wedge R(x) \subseteq W$

$R, M \ \rhd\ \mathbf{fn}_\pi\ x \ \texttt{=>}\ e_0 : S_1 \to S_2 \ \&\ W$
iff $S_1 \sqsubseteq S_2 \wedge \{(m, (\pi, m)) \mid m \in M\} \subseteq W \ \wedge$
$\quad \mathcal{R}_F^{\mathsf{c}}(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \ \rhd\ e_0 : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \ \&\ \mathcal{W}_F(\pi\bullet) \ \wedge$
$\quad R \sqsubseteq \mathcal{R}_F^{\mathsf{d}}(\pi)$

$R, M \ \rhd\ (e_1\ e_2)^l : S_1 \to S_2 \ \&\ W$
iff $R \sqsubseteq R_1 \wedge M \sqsubseteq M_1 \wedge S_1 \sqsubseteq S_{11} \ \wedge$
$\quad R_1, M_1 \ \rhd\ e_1 : S_{11} \to S_{12} \ \&\ W_1 \ \wedge$
$\quad R \sqsubseteq R_2 \wedge M \sqsubseteq M_2 \wedge S_{12} \sqsubseteq S_{21} \ \wedge$
$\quad R_2, M_2 \ \rhd\ e_2 : S_{21} \to S_{22} \ \&\ W_2 \ \wedge$
$\quad \forall \pi \in \{\pi \mid (m, (\pi, m_d)) \in W_1\} :$
$\qquad W_2\lceil X^l_{\mathsf{hc}}(M)\rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge S_{22}\lceil X^l_{\mathsf{hc}}(M)\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \ \wedge$
$\qquad \mathcal{W}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(M)\rceil \subseteq W \wedge \mathcal{S}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(M)\rceil \sqsubseteq S_2 \ \wedge$

$$\mathcal{R}^{\mathsf{d}}_F(\pi)[X^l_{\mathsf{dc}}(M, W_1, \pi)] \sqsubseteq \mathcal{R}^{\mathsf{c}}_F(\pi) \wedge \mathsf{take}_k(l, M) \subseteq \mathcal{M}_F(\pi)$$
for some $R_1, M_1, S_{11}, S_{12}, W_1, R_2, M_2, S_{21}, S_{22}, W_2$

Clearly there will be proposed solutions that are acceptable according to the modified specification but that are not acceptable according to Table 3. This motivates being explicit about what we mean by the best solution. Usually this is taken to mean the smallest solution but this turns out to be "too small" because it allows us to take all solution sets to be the empty ones. To avoid this we shall insist that the *best* solution is the smallest one among all acceptable solutions for which the empty call string ϵ is contained in the set $M_\star$ used for the top level expression $e_\star$. This is still a smallest solution to a specification that has been augmented by the condition $\{\epsilon\} \subseteq M_\star$.

We can now use a result of Tarski [24] to prove that the best solution for one specification equals the best solution for the other. Tarski's result considers a monotone function f on a complete lattice and says that the least fixed point (a fixed point being some v such that $f(v) = v$) equals the least prefixed point (a prefixed point being some v such that $f(v) \sqsubseteq v$). It follows from this result that for monotone functions $f_1, \cdots, f_n$ we have that the least v such that $f_1(v) \sqsubseteq v \wedge \cdots \wedge f_n(v) \sqsubseteq v$ equals the least v such that $f_1(v) \sqcup \cdots \sqcup f_n(v) = v$. In other words, we can change containments to equalities if we "collect" *all* terms defining the same entity.

3 Attribute Grammar Formulations

The flow logic of Table 3 can be transformed into an attribute grammar. The basic idea behind attribute grammars is as follows. Each symbol of the syntax is given a fixed number of attributes with fixed domains. Different instances of the symbols in a syntax tree may have different attribute values. The rules of the syntax are extended with conditions expressing how the attributes of the symbols depend on one another; these conditions have to be fulfilled by the attributes of all instances of the rule in the syntax tree. There are different approaches to the specification of attribute grammars spanning a spectrum from extended attribute grammars [26] that are mainly used for verifying the values of attributes, to the classical attribute grammars (e.g. [27]) that are mainly used for computing the values of attributes.

We shall now proceed in two stages. First we show that a minor transformation will turn the specification of Table 3 into an extended attribute grammar with global attributes and side conditions. The second stage will then transform the extended attribute grammar into an attribute grammar using global attributes and defining the attributes by containments (rather than equalities).

Extended attribute grammars. To specify the extended attribute grammar [26] we shall give the symbol e five *attribute positions* with the following domains:

$$\langle e : R, M, S, S, R(x)\rangle \ ::= \ x$$

$$\begin{array}{l}
\langle e : R, M, S, S, \{(m, (\pi, m)) \mid m \in M\}\rangle \ ::= \\
\quad \mathbf{fn}_\pi \ x \Rightarrow \langle e_0 : \mathcal{R}^{\mathsf{c}}_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi), \mathcal{S}_F(\bullet\pi), \mathcal{S}_F(\pi\bullet), \mathcal{W}_F(\pi\bullet)\rangle \\
\text{with } R \sqsubseteq \mathcal{R}^{\mathsf{d}}_F(\pi)
\end{array}$$

$$\begin{array}{l}
\langle e : R, M, S_1, S_4, W\rangle \ ::= \ (\langle e_1 : R, M, S_1, S_2, W_1\rangle \ \langle e_2 : R, M, S_2, S_3, W_2\rangle)^l \\
\text{with } \forall \pi \in \{\pi \mid (m, (\pi, m_d)) \in W_1\} : \\
\qquad W_2\lceil X^l_{\mathsf{hc}}(M)\rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge S_3\lceil X^l_{\mathsf{hc}}(M)\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \ \wedge \\
\qquad \mathcal{W}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(M)\rceil \subseteq W \wedge \mathcal{S}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(M)\rceil \sqsubseteq S_4 \ \wedge \\
\qquad \mathcal{R}^{\mathsf{d}}_F(\pi)\lceil X^l_{\mathsf{dc}}(M, W_1, \pi)\rceil \sqsubseteq \mathcal{R}^{\mathsf{c}}_F(\pi) \wedge \mathsf{take}_k(l, M) \subseteq \mathcal{M}_F(\pi)
\end{array}$$

Table 4. Extended attribute grammar formulation.

- $\widehat{\mathsf{Env}}$, $\mathcal{P}(\mathsf{Mem})$, $\widehat{\mathsf{Store}}$, $\widehat{\mathsf{Store}}$ and $\widehat{\mathsf{Val}}$.

In the notation of extended attribute grammars the symbol e and its attributes are written as

$$\langle e : R, M, S_1, S_2, W\rangle$$

Here R, M, S_1, S_2 and W are *terms* defining the attributes of the corresponding position; these terms are constructed from constant values, attribute variables and various operations on terms. Multiple occurrences of the same attribute variable in the same syntactic rule expresses an *implicit* condition since in each instance of the rule the occurrences must have the same value.

In addition to the attributes associated with e we shall use five global attributes corresponding to the five caches of Table 2:

- $\mathcal{R}^{\mathsf{d}}_F$, $\mathcal{R}^{\mathsf{c}}_F$, $\mathcal{M}_F$, $\mathcal{S}_F$ and $\mathcal{W}_F$.

The global attributes can be used as constants in the construction of terms for the attributes and their values can be further constrained by explicit conditions associated with the syntactic rules.

It is now easy to see that Table 4 can be obtained from Table 3 and vice versa by simply changing the notation. Hence it should be clear that the two specifications admit the same acceptable solutions and therefore that the best solution for one equals the best solution for the other.

Remark. We should point out that the specification of Table 4 goes a little beyond the extended attribute grammars of [26] in that it uses global attributes and explicit conditions. This can be rectified using standard transformation techniques: for each global attribute we can give e an additional attribute position and we can also give e a single attribute position holding the conjunction of the explicit conditions. □

Attribute grammars. So far we have used extended attribute grammars as a *verification mechanism*: the (implicit and explicit) conditions associated with the syntactic rules specify a relationship between the values of the (local and global) attributes that have to hold; they do not directly specify how the attributes have to be computed from one another. Extended attribute grammars can also be given a more computational interpretation and when doing so it is customary to introduce a distinction between *inherited* and *synthesised* attributes: the idea is that the inherited attributes will carry information from the root of the syntax tree towards its leaves whereas the synthesised attributes will carry information in the opposite direction.

In the case of attribute grammars [27] we formalise the ideas as follows. The symbol e is equipped with the following named attributes:

- inherited attributes R, M and S_1 with domains $\widehat{\mathsf{Env}}$, $\mathcal{P}(\mathsf{Mem})$, and $\widehat{\mathsf{Store}}$ respectively, and
- synthesised attributes S_2 and W with domains $\widehat{\mathsf{Store}}$ and $\widehat{\mathsf{Val}}$ respectively.

In this way information about the environment (R), the context (M) and the store (S_1) in which the expression is evaluated will be given by the inherited attributes, whereas information about the store (S_2) and value (W) obtained as a result of evaluating the expression will be given by the synthesised attributes. The values of the attributes are referenced using dot-notation (for example $e.\mathsf{R}$) and they are specified by explicit conditions associated with the syntactic rules.

The computational nature of the attribute grammar notation is further emphasised by the distinction between defining and applied attributes of a syntactic rule: a *defining* attribute is either an inherited attribute of the left hand side or it is a synthesised attribute of one of the symbols on the right hand side, whereas an *applied* attribute either is a synthesised attribute of the left hand side or an inherited attribute of one of the symbols on the right hand side. The conditions associated with the syntactic rule usually specify how to compute the values of the applied attributes from those of the defining attributes.

In Table 5 we give an algorithm for transforming an extended attribute grammar into an attribute grammar. The idea is as follows. Step 1 takes care of the fact that positions corresponding to defining attributes typically are given by terms and not just attribute variables; this expresses an implicit condition that now is made explicit. Step 2 ensures that the implicit condition expressed by using the same attribute variable in different defining positions is made explicit. After these two steps all defining attribute positions contain distinct attribute variables and in step 3 we rename them to use the dot notation. In step 4 we take care of the applied attributes and replace them by their name and an explicit condition for their computation. In step 5 we attempt to remove any attribute variables that still occur, and finally, in step 6 we change the notation to be that of attribute grammars [27]. This transformation procedure is more complex than the one described in [26, 14] because we need to deal with containments and we take care not to generate more containments than absolutely necessary.

Step 1: For all positions corresponding to defining attributes with a term U that is not just an attribute variable do the following:
- replace the term with a fresh attribute variable u, and
- add the side condition $u \sqsubseteq U$ if the position is on the right hand side, and $U \sqsubseteq u$ if it is on the left hand side.

Step 2: If the same attribute variable u occurs in more than one position corresponding to a defining attribute then do the following:
- replace each of the positions with a fresh attribute variable u_i, and
- add the condition $u_i \sqsubseteq u$ if the position is on the right hand side, and $u \sqsubseteq u_i$ if it is on the left hand side.

Step 3: For all defining attributes do the following:
- if the attribute variable u is in the position for the attribute U of the symbol e then replace all occurrences of u with $e.\mathsf{U}$.

Step 4: For all applied attributes do the following:
- if the term U is in the position for the attribute U of the symbol e then replace it with $e.\mathsf{U}$ and add the condition $U \sqsubseteq e.\mathsf{U}$.

Step 5: Simplify the conditions (during which any remaining attribute variables are regarded as being existentially quantified).

Step 6: Rewrite the rule using attribute grammar notation.

Table 5. From extended attribute grammars to attribute grammars.

Using this algorithm on the specification in Table 4 results in the specification in Table 6. It is interesting to note that Figure 1 may now be viewed as illustrating the dependency graphs for function application and function definition.

In analogy with the discussion about "containments versus equalities" (in the previous section) it will not be the case that the specifications in Tables 4 and 6 admit the same acceptable solutions. However, it will be the case that the best solution for one equals the best solution for the other. This will suffice for our purposes because program analysis is concerned with obtaining the best solution to a given specification.

It is easy to see that the specification in Table 6 is in fact circular. Consider an expression $\texttt{fn}_\pi\ x$ `=>` e where e is an application $(e_1\ e_2)^l$. Then the clause for function definition gives

$$e.\mathsf{W} \subseteq \mathcal{W}_F(\pi\bullet)$$

and the clause for application gives

$$\mathcal{W}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(e.\mathsf{M})\rceil \subseteq e.\mathsf{W}$$

showing that $\mathcal{W}_F(\pi\bullet)$ depends on $e.\mathsf{W}$ which in turn depends on $\mathcal{W}_F(\pi\bullet)$.

Remark. The specification of Table 6 is not quite an attribute grammar in the classical sense [27]: it uses quantifiers, it uses global attributes, and it uses containments rather than equalities. The quantified formula can be eliminated by

$e ::= x$	$e ::= (e_1\ e_2)^l$
$\quad e.\mathsf{S}_1 \sqsubseteq e.\mathsf{S}_2$	$\quad e.\mathsf{R} \sqsubseteq e_1.\mathsf{R}$
$\quad e.\mathsf{R}(x) \subseteq e.\mathsf{W}$	$\quad e.\mathsf{M} \subseteq e_1.\mathsf{M}$
	$\quad e.\mathsf{S}_1 \sqsubseteq e_1.\mathsf{S}_1$
$e ::= \mathbf{fn}_\pi\ x \texttt{ => } e_0$	$\quad e.\mathsf{R} \sqsubseteq e_2.\mathsf{R}$
$\quad e.\mathsf{S}_1 \sqsubseteq e.\mathsf{S}_2$	$\quad e.\mathsf{M} \subseteq e_2.\mathsf{M}$
$\quad \{(m,(\pi,m)) \mid m \in e.\mathsf{M}\} \subseteq e.\mathsf{W}$	$\quad e_1.\mathsf{S}_2 \sqsubseteq e_2.\mathsf{S}_1$
$\quad \mathcal{R}^{\mathsf{c}}_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)] \sqsubseteq e_0.\mathsf{R}$	$\quad \forall\pi \in \{\pi \mid (m,(\pi,m_d)) \in e_1.\mathsf{W}\} :$
$\quad \mathcal{M}_F(\pi) \subseteq e_0.\mathsf{M}$	$\qquad e_2.\mathsf{W}\lceil X^l_{\mathsf{hc}}(e.\mathsf{M})\rceil \subseteq \mathcal{W}_F(\bullet\pi)$
$\quad \mathcal{S}_F(\bullet\pi) \sqsubseteq e_0.\mathsf{S}_1$	$\qquad e_2.\mathsf{S}_2\lceil X^l_{\mathsf{hc}}(e.\mathsf{M})\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi)$
$\quad e_0.\mathsf{S}_2 \sqsubseteq \mathcal{S}_F(\pi\bullet)$	$\qquad \mathcal{W}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(e.\mathsf{M})\rceil \subseteq e.\mathsf{W}$
$\quad e_0.\mathsf{W} \subseteq \mathcal{W}_F(\pi\bullet)$	$\qquad \mathcal{S}_F(\pi\bullet)\lceil X^l_{\mathsf{ch}}(e.\mathsf{M})\rceil \sqsubseteq e.\mathsf{S}_2$
$\quad e.\mathsf{R} \sqsubseteq \mathcal{R}^{\mathsf{d}}_F(\pi)$	$\qquad \mathcal{R}^{\mathsf{d}}_F(\pi)\lceil X^l_{\mathsf{dc}}(e.\mathsf{M}, e_1.\mathsf{W}, \pi)\rceil \sqsubseteq \mathcal{R}^{\mathsf{c}}_F(\pi)$
	$\qquad \mathsf{take}_k(l, e.\mathsf{M}) \subseteq \mathcal{M}_F(\pi)$

Table 6. Attribute grammar formulation.

collecting left hand sides and by expressing the constraints at the level of complete caches rather than at the level of individual entries into caches. Global attributes can be eliminated by replacing them with pairs of inherited and synthesised attributes that are threaded through the syntax tree. Finally, the containments can be replaced by equalities provided that all the relevant containments are "collected" as explained in Section 2. □

4 Verbose Flow Logics

The (somewhat liberal) attribute grammar formulation obtained in Table 6 goes a long way towards an efficient implementation of the flow logic. However, compared with the efficient constraint based methods it still suffers the disadvantage that it depends too closely on the program syntax and hence makes it hard for an implementation to avoid the penalty of being syntax-directed. To overcome this problem we shall introduce additional caches for representing the attributes instead of using the dot-notation.

Additional caches. To facilitate introducing the new caches we need to demand that *all* subexpressions are labelled; previously only the applications were labelled. One way to achieve this is to differentiate between labelled expressions and unlabelled terms:

$$
\begin{aligned}
e &::= t^l \\
t &::= x \mid \mathbf{fn}_\pi\ x \texttt{ => } e_0 \mid (e_1\ e_2) \mid \cdots
\end{aligned}
$$

We may think of the labels as tree addresses or names of the nodes in the syntax tree. As before it is the expressions that will be analysed.

The new caches are defined in Table 7 and they can be summarised as follows:

$$\begin{aligned}
\mathcal{R}_L &\in \widehat{\mathsf{RCache}}_{\mathsf{L}} = \mathsf{Lab} \to \widehat{\mathsf{Env}} \\
\mathcal{M}_L &\in \widehat{\mathsf{MCache}}_{\mathsf{L}} = \mathsf{Lab} \to \mathcal{P}(\mathsf{Mem}) \\
\mathcal{W}_L &\in \widehat{\mathsf{WCache}}_{\mathsf{L}} = \mathsf{Lab} \to \widehat{\mathsf{Val}} \\
\mathcal{S}_L &\in \widehat{\mathsf{SCache}}_{\mathsf{L}} = (\bullet\mathsf{Lab} \cup \mathsf{Lab}\bullet) \to \widehat{\mathsf{Store}}
\end{aligned}$$

Table 7. Additional caches.

- The *environment cache* $\mathcal{R}_L$ corresponds to the R attribute: we shall replace $t^l.\mathsf{R}$ by $\mathcal{R}_L(l)$.
- The *memento cache* $\mathcal{M}_L$ corresponds to the M attribute: we shall replace $t^l.\mathsf{M}$ by $\mathcal{M}_L(l)$.
- The *value cache* $\mathcal{W}_L$ corresponds to the W attribute: we shall replace $t^l.\mathsf{W}$ by $\mathcal{W}_L(l)$.
- The *store cache* $\mathcal{S}_L$ corresponds to the S_1 and S_2 attributes: we shall replace $t^l.\mathsf{S}_1$ by $\mathcal{S}_L(\bullet l)$ and $t^l.\mathsf{S}_2$ by $\mathcal{S}_L(l\bullet)$.

Control flow formulation. The description of the caches already suggests a method for transforming the specification of Table 6. In doing so we shall exploit the presence of labels on all subexpressions. We shall write the analysis of an expression t^l as

$$(\mathcal{R}_F^{\mathsf{d}}, \mathcal{R}_F^{\mathsf{c}}, \mathcal{M}_F, \mathcal{S}_F, \mathcal{W}_F, \mathcal{R}_L, \mathcal{M}_L, \mathcal{W}_L, \mathcal{S}_L) \text{ satisfies} \rhd t^l$$

and (as in Table 3) we shall be explicit about the analysis of subexpressions. Allowing minor changes in notation this results in the specification of Table 8.

The new formulation is typical of the abstract control flow specification of [16, 6, 9] except that it is syntax-directed and therefore closer to implementation. If all terms are uniquely labelled then the specification in Table 8 can be converted back into the one in Table 6. Hence the two specifications admit the same acceptable solutions and it follows that the best solution for one equals the best solution for the other.

Constraint generation. The final phase in implementing the flow logic consists in changing the specification from being a recipe for verifying the acceptability of a proposed solution to being a method for computing the best solution. This involves extracting the individual conjuncts of the specification and we shall do so by defining a function C that maps an expression to the set of (possibly conditional) constraints involved in verifying the acceptability of a proposed solution. In the case of variables and function definition this is rather straightforward as illustrated in Table 9. In the case of function application there is a small complication. Writing $\mathsf{FUN}(W) = \{\pi \mid (m, (\pi, m_d)) \in W\}$ we have to generate constraints for each $\pi \in \mathsf{FUN}(\mathcal{W}_L(l_1))$ and the problem is that the value of $\mathcal{W}_L(l_1)$ is not known at constraint generation time. Instead we generate *conditional constraints* for each π occurring in the program (assuming that the program is a closed system); in this way the decision whether or not to really impose the constraint is delayed until a later point in time where information about the value of $\mathcal{W}_L(l_1)$ is available.

$$\begin{array}{l}
\rhd\ x^{l} \\
\text{iff}\ \ \mathcal{S}_L(\bullet l) \sqsubseteq \mathcal{S}_L(l\bullet)\ \wedge \\
\quad\ \mathcal{R}_L(l)(x) \subseteq \mathcal{W}_L(l) \\
\\
\rhd\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ t_0^{l_0})^{l} \\
\text{iff}\ \ \mathcal{S}_L(\bullet l) \sqsubseteq \mathcal{S}_L(l\bullet) \wedge \\
\quad \{(m,(\pi,m)) \mid m \in \mathcal{M}_L(l)\} \\
\qquad \subseteq \mathcal{W}_L(l) \wedge \\
\quad \mathcal{R}_F^{\mathsf{c}}(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)] \\
\qquad \sqsubseteq \mathcal{R}_L(l_0) \wedge \\
\quad \mathcal{M}_F(\pi) \subseteq \mathcal{M}_L(l_0) \wedge \\
\quad \mathcal{S}_F(\bullet\pi) \sqsubseteq \mathcal{S}_L(\bullet l_0) \wedge \\
\quad \rhd\ t_0^{l_0} \wedge \\
\quad \mathcal{S}_L(l_0\bullet) \sqsubseteq \mathcal{S}_F(\pi\bullet) \wedge \\
\quad \mathcal{W}_L(l_0) \subseteq \mathcal{W}_F(\pi\bullet) \wedge \\
\quad \mathcal{R}_L(l) \sqsubseteq \mathcal{R}_F^{\mathsf{d}}(\pi)
\end{array}$$

$$\begin{array}{l}
\rhd\ (t_1^{l_1}\ t_2^{l_2})^{l} \\
\text{iff}\ \ \mathcal{R}_L(l) \sqsubseteq \mathcal{R}_L(l_1) \wedge \\
\quad \mathcal{M}_L(l) \subseteq \mathcal{M}_L(l_1) \wedge \\
\quad \mathcal{S}_L(\bullet l) \sqsubseteq \mathcal{S}_L(\bullet l_1) \wedge \\
\quad \rhd\ t_1^{l_1} \wedge \\
\quad \mathcal{R}_L(l) \sqsubseteq \mathcal{R}_L(l_2) \wedge \\
\quad \mathcal{M}_L(l) \subseteq \mathcal{M}_L(l_2) \wedge \\
\quad \mathcal{S}_L(l_1\bullet) \sqsubseteq \mathcal{S}_L(\bullet l_2) \wedge \\
\quad \rhd\ t_2^{l_2} \wedge \\
\quad \forall \pi \in \{\pi \mid (m,(\pi,m_d)) \in \mathcal{W}_L(l_1)\} : \\
\qquad \mathcal{W}_L(l_2)\lceil X_{\mathsf{hc}}^{l}(\mathcal{M}_L(l))\rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge \\
\qquad \mathcal{S}_L(l_2)\lceil X_{\mathsf{hc}}^{l}(\mathcal{M}_L(l))\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \wedge \\
\qquad \mathcal{W}_F(\pi\bullet)\lceil X_{\mathsf{ch}}^{l}(\mathcal{M}_L(l))\rceil \subseteq \mathcal{W}_L(l) \wedge \\
\qquad \mathcal{S}_F(\pi\bullet)\lceil X_{\mathsf{ch}}^{l}(\mathcal{M}_L(l))\rceil \sqsubseteq \mathcal{S}_L(l\bullet) \wedge \\
\qquad \mathcal{R}_F^{\mathsf{d}}(\pi)\lceil X_{\mathsf{dc}}^{l}(\mathcal{M}_L(l),\mathcal{W}_L(l_1),\pi)\rceil \sqsubseteq \mathcal{R}_F^{\mathsf{c}}(\pi) \wedge \\
\qquad \mathsf{take}_k(l,\mathcal{M}_L(l)) \subseteq \mathcal{M}_F(\pi)
\end{array}$$

Table 8. Control flow formulation: verbose flow logic.

It is immediate that the two specifications admit the same acceptable solutions and hence that the best solution for one equals the best solution for the other.

Constraint solving. To compute the best solution to the constraints generated we shall construct a graph and then propagate information until stabilisation. Each entry in each cache gives rise to a node; with each node is associated a data entry for holding the corresponding value. Each constraint may give rise to one or more edges; they are represented using a constraint list for each node that contains the outgoing edges for that node. The algorithm operates using a worklist that contains those nodes for which we still might need to propagate information along their outgoing edges. Since we wish to compute the best (rather than merely the smallest solution) the worklist is initially set to $\mathcal{M}_L(l_\star)$ where $l_\star$ is the label of the top level expression $e_\star$; also the data entry for $\mathcal{M}_L(l_\star)$ is initially set to $\{\epsilon\}$ (as discussed in Section 2). The details of the algorithm are given in Table 10 and are mostly quite standard (e.g. [6]).

One can prove that the algorithm always terminates and that it produces the best solution to the constraints upon which it operates; it is essential for this that *all* constraints generated by Table 9 are of the form $P \sqsubseteq p$ or $\pi \in P_1 \Rightarrow P_2 \sqsubseteq p$ where p is a node (as opposed to a more general expression or a constant set) and where P, P_1 and P_2 all contain at least one node. The efficiency of the algorithm can be improved in many ways [4] but in the special case of functional programs without side effects the formulation given in Table 10 suffices for achieving the best known (cubic) worst case time complexity.

The above procedure is an exhaustive algorithm [1] for solving the constraints. By contrast the extended attribute grammar evaluation scheme described in [14] for mainly non-circular extended attribute grammars is demand driven [2]. To obtain a demand driven procedure for solving the circular constraints one would probably need to exploit ideas from minimal function graphs [11].

$$\mathcal{C}\llbracket x^l \rrbracket = \{\mathcal{S}_L(\bullet l) \sqsubseteq \mathcal{S}_L(l\bullet), \mathcal{R}_L(l)(x) \subseteq \mathcal{W}_L(l)\}$$

$$\begin{array}{l}
\mathcal{C}\llbracket (\mathtt{fn}_\pi\ x \mathtt{=>}\ t_0^{l_0})^l \rrbracket \\
\ = \{\mathcal{S}_L(\bullet l) \sqsubseteq \mathcal{S}_L(l\bullet), \{(m,(\pi,m)) \mid m \in \mathcal{M}_L(l)\} \subseteq \mathcal{W}_L(l), \\
\quad \mathcal{R}_F^{\mathsf{c}}(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)] \sqsubseteq \mathcal{R}_L(l_0), \mathcal{M}_F(\pi) \subseteq \mathcal{M}_L(l_0), \mathcal{S}_F(\bullet\pi) \sqsubseteq \mathcal{S}_L(\bullet l_0)\} \\
\quad \cup\ \mathcal{C}\llbracket t_0^{l_0} \rrbracket \cup \\
\quad \{\mathcal{S}_L(l_0\bullet) \sqsubseteq \mathcal{S}_F(\pi\bullet), \mathcal{W}_L(l_0) \subseteq \mathcal{W}_F(\pi\bullet), \mathcal{R}_L(l) \sqsubseteq \mathcal{R}_F^{\mathsf{d}}(\pi)\}
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\llbracket (t_1^{l_1}\ t_2^{l_2})^l \rrbracket \\
\ = \{\mathcal{R}_L(l) \sqsubseteq \mathcal{R}_L(l_1), \mathcal{M}_L(l) \subseteq \mathcal{M}_L(l_1), \mathcal{S}_L(\bullet l) \sqsubseteq \mathcal{S}_L(\bullet l_1)\} \\
\quad \cup\ \mathcal{C}\llbracket t_1^{l_1} \rrbracket \cup \\
\quad \{\mathcal{R}_L(l) \sqsubseteq \mathcal{R}_L(l_2), \mathcal{M}_L(l) \subseteq \mathcal{M}_L(l_2), \mathcal{S}_L(l_1\bullet) \sqsubseteq \mathcal{S}_L(\bullet l_2)\} \\
\quad \cup\ \mathcal{C}\llbracket t_2^{l_2} \rrbracket \cup \\
\quad \{\pi \in \mathsf{FUN}(\mathcal{W}_L(l_1)) \Rightarrow \mathcal{W}_L(l_2)\lceil X_{\mathsf{hc}}^l(\mathcal{M}_L(l))\rceil \subseteq \mathcal{W}_F(\bullet\pi), \\
\quad\ \ \pi \in \mathsf{FUN}(\mathcal{W}_L(l_1)) \Rightarrow \mathcal{S}_L(l_2)\lceil X_{\mathsf{hc}}^l(\mathcal{M}_L(l))\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi), \\
\quad\ \ \pi \in \mathsf{FUN}(\mathcal{W}_L(l_1)) \Rightarrow\ \mathcal{W}_F(\pi\bullet)\lceil X_{\mathsf{ch}}^l(\mathcal{M}_L(l))\rceil \subseteq \mathcal{W}_L(l), \\
\quad\ \ \pi \in \mathsf{FUN}(\mathcal{W}_L(l_1)) \Rightarrow \mathcal{S}_F(\pi\bullet)\lceil X_{\mathsf{ch}}^l(\mathcal{M}_L(l))\rceil \sqsubseteq \mathcal{S}_L(l\bullet) \\
\quad\ \ \pi \in \mathsf{FUN}(\mathcal{W}_L(l_1)) \Rightarrow \mathcal{R}_F^{\mathsf{d}}(\pi)\lceil X_{\mathsf{dc}}^l(\mathcal{M}_L(l), \mathcal{W}_L(l_1), \pi)\rceil \sqsubseteq \mathcal{R}_F^{\mathsf{c}}(\pi), \\
\quad\ \ \pi \in \mathsf{FUN}(\mathcal{W}_L(l_1)) \Rightarrow \mathsf{take}_k(l, \mathcal{M}_L(l)) \subseteq \mathcal{M}_F(\pi), \\
\qquad\qquad \mid \pi \text{ occurs in the program}\}
\end{array}$$

Table 9. Constraint generation.

5 Conclusion

The literature contains a number of seemingly different approaches to the specification of program analyses. The main raison d'être for *flow logic* is that it is sufficiently compact and sufficiently unbiased that it facilitates incorporating insights from many different approaches: the full specification [17] integrates insights from control flow analysis of functional languages [21, 9] with insights from interprocedural data flow analysis of object-oriented languages [19]. In fact, flow logics are unbiased as regards the choice of language paradigms, program properties, kinds of formal semantics, and methods used for computing the best solution. The latter point is achieved by clearly separating *verifying* the acceptability of a proposed solution from *computing* the best solution and we believe that flow logic does so in a way that is much closer to the field of program analysis than previous attempts at formulating program analysis in logical form (e.g. [12, 3, 10]).

This paper has demonstrated one possible route for implementing a compositional *flow logic* in *succinct* form. We have argued that the flow logic is closely related to a specification using *extended attribute grammars* with global attributes and side conditions. A key step concerned transforming the extended attribute grammar into a circular *attribute grammar* using global attributes and where the attributes are defined using containments (or inclusions) rather than equalities. The attribute grammar specification is implicit about the nodes of the syntax trees and by using labels to make them explicit we arrive at a specification in the *constraint based formulation* often used for control flow analysis and set based

```
Step 1: Initialise the data structures:
    • the initially empty worklist is initialised as follows:
      – $\mathcal{M}_L(l_\star)$ is included in the worklist
    • the initially empty data entries are initialised as follows:
      – $\{\epsilon\}$ is included in the data entry for $\mathcal{M}_L(l_\star)$
    • the initially empty constraint lists are initialised as follows:
      – a constraint of the form $P \sqsubseteq p$ is included in the constraint list of
        all nodes occurring in $P$
      – a constraint of the form $\pi \in P_1 \Rightarrow P_2 \sqsubseteq p$ is included in the
        constraint lists of all nodes occurring in $P_1$ or $P_2$
Step 2: Repeat the following until the worklist is empty:
    • remove some node $q$ from the worklist
    • for each constraint of the form $P \sqsubseteq p$ in the constraint list for $q$ do
      the following:
      – if the current value of $P$ contains one or more elements that are
        are not already present in the data entry for $p$ then add them and
        include $p$ in the worklist
    • for each constraint of the form $\pi \in P_1 \Rightarrow P_2 \sqsubseteq p$ in the constraint
      list for $q$ do the following:
      – if the current value of $P_1$ contains $\pi$ and if the current value of $P_2$
        contains one or more elements that are not already present in the
        data entry for $p$ then add them and include $p$ in the worklist
```

Table 10. Solving the constraints.

analysis (and that may be regarded as a flow logic in *verbose* form). The constraint based specification is then easily modified so as to *generate constraints* that are subsequently solved using a graph based algorithm; only in the very final stage do we commit ourselves to performing exhaustive analysis as opposed to demand analysis which is equally possible. Indeed, much of the flexibility of flow logic stems from its ability to be implemented in more than one way so as to suit the demands of the application.

Acknowledgement. This work has been supported in part by the DART project funded by the Danish Science Research Council. Special thanks are due to Reinhard Wilhelm: this paper would never have come into existence without his provoking remarks!

References

1. W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis — Part I. Exhaustive analysis. *Acta Informatica*, 10:245–264, 1978.
2. W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis — Part II. Demand analysis. *Acta Informatica*, 10:265–272, 1978.
3. P. N. Benton. Strictness logic and polymorphic invariance. In *Proc. Second International Symposium on Logical Foundations of Computer Science*, pages 33–44. Springer Lecture Notes in Computer Science 620, 1992.

4. C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proc. SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 189–204. Springer-Verlag, 1996.
5. Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT'89*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1989.
6. K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of ICFP'97*, pages 38–51. ACM Press, 1997.
7. M. S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, 1977.
8. S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proc. LFP'94*, pages 294–305, 1994.
9. S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95.* ACM Press, 1995.
10. T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, 1993.
11. N .D. Jones and A. Mycroft. Dataflow of Applicative Programs Using Minimal Function Graphs. In *Proc. 13th POPL*, pages 296–306. ACM Press, 1986.
12. T. M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. FPCA '89*, pages 260–272. ACM Press, 1989.
13. W. Landi and B. G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *18th POPL, Orlando, Florida*, pages 93–103. ACM Press, 1991.
14. O. L. Madsen. On defining semantics by means of extended attribute grammars. In N. D. Jones, editor, *Proc. Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 259–299. Springer-Verlag, 1980.
15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.
16. F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL '97*, 1997.
17. F. Nielson and H. R. Nielson. Interprocedural Flow Logics. Manuscript, 1998.
18. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis: Flows and Effects.* To appear, 1999.
19. H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. In *Proc. SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, 1996.
20. 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, 1981.
21. O. Shivers. The semantics of Scheme control-flow analysis. In *Partial Evaluation and Semantics-Based Program Manipulation.* ACM SIGPLAN Notices 26 (9), 1991.
22. G. S. Smith. Polymorphic type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.
23. J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
24. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.
25. J. Vitek, R. N. Horspool, and J. S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer-Verlag, 1992.
26. D. A. Watt and O. L. Madsen. Extended attribute grammars. *Computer Journal*, 26(2):142–153, 1983.
27. R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley, 1995.