Systematic Realisation of Control Flow Analyses for CML

Kirsten L. Solberg Gasser, Flemming Nielson, Hanne Riis Nielson

Computer Science Department, Aarhus University,
Ny Munkegade, DK-8000 Aarhus C, Denmark

Electronic mail: {kls,fn,hrn}@daimi.aau.dk

Abstract We present a methodology for the systematic realisation of control flow analyses and illustrate it for Concurrent ML. We start with an abstract specification of the analysis that is next proved semantically sound with respect to a traditional small-step operational semantics; this result holds for terminating as well as non-terminating programs. The analysis is defined coinductively and it is shown that all programs have a least analysis result (that is indeed the best one). To realise the analysis we massage the specification in three stages: (i) to explicitly record reachability of subexpressions, (ii) to be defined in a syntax-directed manner, and (iii) to generate a set of constraints that subsequently can be solved by standard techniques. We prove equivalence results between the different versions of the analysis; in particular it follows that the least solution to the constraints generated will be the least analysis result also to the initial specification.

1 Introduction

Many compiler optimisation techniques rely on control flow information: for a given program point, which program points can the flow of control jump to? For imperative programs without procedures this question may be simple to answer but for more powerful languages, whether imperative languages with procedures, functional languages, concurrent languages, or object-oriented languages, it is much more complicated. In the literature quite some effort has been devoted to the development of control flow analyses for functional languages, see e.g. [1, 5, 8, 9, 10, 13, 15, 18, 19, 20, 21].

We believe that the overall development of control flow analyses should follow the core methodology of abstract interpretation [2, 3, 4, 11]:

\[ \text{sem} \leadsto \text{po}_0 \leadsto \cdots \leadsto \text{po}_n \leadsto \text{impl} \]

Given a programming language and its semantics (\text{sem}), we consider an abstract analysis (\text{po}_0) and show it to be semantically correct. We then systematically massage the analyses in such a way that (i) semantic correctness is maintained, and (ii) we end up with an analysis (\text{po}_n) that may be implemented efficiently (\text{impl}). The core methodology also opens up for the possibility of counselling the analyses (using Galois connections, widening operators, and narrowing operators) but we are not going to explore this in the present paper.

We shall illustrate the methodology by developing a control flow analysis for Concurrent ML (CML) [17]. CML is an extension of the functional language Standard ML (SML) [12] with primitives for the dynamic creation of processes and channels and for the synchronous communication of values over channels. Channels as well as functions are first class values and thus they can be supplied to functions as parameters, returned as results, and transmitted over channels. Compared with traditional functional languages, the control flow of CML programs is further complicated by the fact that

- closures may be communicated over channels and hence invoked on other processes than where they are created, and
- channels may be communicated over channels and later used for new communications.

The analysis we present in this paper is an extension of the traditional 0-CFA analyses for functional languages to the concurrency primitives of CML.

Relationship to other work In the literature several control flow analyses have been developed for functional languages; some using the standard syntax for such languages and others using an intermediate language (like continuation passing style or "\lambda-normal form") of a specific implementation. The analyses have been specified in different styles and even if we restrict attention to those lending themselves naturally to a constraint based formalisation the variation is surprisingly large. In one end of the spectrum the specifications are rather abstract; constraints are only specified implicitly and subexpressions may be analysed several times depending on the context in which they arise. In the other end of the spectrum we have analyses that are specified at a level much closer to the actual realisation: they are explicit about the generation of constraints and they proceed in a syntax-directed manner such that subexpressions are analysed at most once. Only very recently [33] it was pointed out that the first kind of analyses should be defined conductively (and not inductively) in order to make sense; the second kind of analyses are naturally defined inductively. We believe the present paper is the first to explore the formal relationship between the two styles of specification.

Control flow analysis of concurrent languages have only received rather little attention: Jagannathan and Weeks [8] study a lambda calculus extended with a spawn construct
for creating parallel threads that communicate via first class shared locations, and they present a 0-CFA like analysis with the additional twist that a 1-CFA like approach is used to distinguish processes. Flanagan and Felleisen [8] study a lambda calculus with a 

\[ \text{fork } e \] creates a new process; the expression evaluated is the expression \( e \).

\[ \text{channel } e \] creates a new channel; the expression evaluated is the expression \( e \).

\[ \text{send } e_1, e_2 \] transmits the value of \( e_2 \) on the channel that \( e_1 \) evaluates to (provided that another process will engage in the communication).

\[ \text{receive } e \] accepts a value on the channel that \( e \) evaluates to (provided that another process will engage in the communication).

The next step towards realisation of the analysis is to turn it into a syntax-directed specification. To do that we first note that the abstract analysis may analyse the same program fragments several times but (unlike type systems) it is only concerned with the reachable program fragments. As a first step towards the syntax-directed analysis, Section 4 reformulates the abstract analysis to explicitly keep track of the reachable program fragments although they may still be analysed several times. The resulting reachability closure analysis is also defined as the greatest fixed point of a function, and a coinductive proof shows that it admits the same analyses as the abstract closure analysis of Section 3, and that it enjoys a Moore family property. This means that the least analysis result of the modified analysis also is the least analysis result of the original one and vice versa.

In Section 5 we present the syntax-directed closure analysis. It incorporates the reachability aspect of the previous analysis but in such a way that each program fragment is analysed at most once. Since this analysis is syntax-directed we can safely define it as the least fixed point of the specification as indeed it will only have one fixed point. We show that the analysis results obtainable by the syntax-directed specification also are obtainable in the reachability specification and furthermore, that the least (or best) analysis result that can be obtained by the reachability specification equals the one obtained by the syntax-directed specification.

The syntax-directed specification can then be turned into a syntax-directed function \( \mathcal{C} \) for collecting constraint. We illustrate this in Section 6 where we also show that the constraints generated have the same solutions as the syntax-directed specification. Standard constraint solving algorithms can now be employed to find the least solution to the constraints, and we briefly sketch an \( O(n^3) \) algorithm for this (where \( n \) is the size of the program). Composing the correctness results then yields an implementation that will compute the least analysis result with respect to the abstract specification of the analysis and we know that it will be semantically correct.

Finally, Section 7 contains the concluding remarks. The Appendix contains the non-trivial parts of the proofs of the main results of the paper but can safely be omitted; it also contains details of the constraint solving technique.

2 Concurrent ML

Concurrent ML is an extension of SML with concurrency primitives for synchronous communication over channels. We shall be interested in the following operations:

\[ \text{fork } e \] creates a new process; the expression evaluated is the expression \( e \).

\[ \text{channel } e \] creates a new channel; the expression evaluated is the expression \( e \).

\[ \text{send } e_1, e_2 \] transmits the value of \( e_2 \) on the channel that \( e_1 \) evaluates to (provided that another process will engage in the communication).

\[ \text{receive } e \] accepts a value on the channel that \( e \) evaluates to (provided that another process will engage in the communication).

We shall consider a subset of CML that additionally has explicit operators for conditional, recursion and local definitions. Note that \textit{send} and \textit{receive} take care of their own synchronisation; hence we can dispense with the \textit{sync} operation. For the presentation of our analysis it is important that we are able to label all program fragments and we therefore present the syntax using expressions and terms:

\[ e \in \text{Exp} \quad \text{(expressions, i.e. labelled terms)} \]

\[ e ::= t \]

\[ t \in \text{Term} \quad \text{(terms, i.e. unlabelled expressions)} \]

\[ t ::= e \mid x \mid \text{fn } x \Rightarrow e_0 \mid \text{fun } f x \Rightarrow e_0 \mid e_1, e_2 \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{fork } e \mid \text{channel } e \mid \text{send } e_1, e_2 \mid \text{receive } e \]

\[ l \in \text{Lab} \quad \text{(labels)} \]

\[ c \in \text{Const} \quad \text{(constants)} \]

\[ x \in \text{Var} \quad \text{(variables)} \]

Here \( \text{fn } x \Rightarrow e \) is the function abstraction, \( \text{fun } f x \Rightarrow e \) is a recursive variant of \( \text{fn } x \Rightarrow e \) where all free occurrences of \( f \) in \( e \) refer to \( \text{fun } f x \Rightarrow e \) itself, and \( \text{let } x = e_1 \text{ in } e_2 \) is a non-recursive local definition that is equivalent to (fn
$x \mapsto e_2(e_1)$. As usual we shall use parentheses to disambiguate the parsing whenever needed. Also we shall assume throughout that all occurrences of fun $f \mapsto e$ have $f$ and $x$ to be distinct. For simplicity of presentation we shall assume that there are no functional constants; however, it would be straightforward to add a syntactic clause like "$\ast := e_1 op e_2$" for a class op of binary operators (like addition).

We shall equip this language with a small-step operational semantics using environments in the style proposed by Plotkin [16]. This necessitates augmenting the syntax with notation for closures and for local environments. We therefore define intermediate expressions and terms as follows:

$$
\begin{align*}
\text{ie} & \in IExp \quad \text{(intermediate expressions)} \\
\text{ie} := & \text{id}' \\
\text{it} & \in ITerm \quad \text{(intermediate terms)} \\
\text{it} := & c \mid \text{fn } x \mapsto e_0 \mid \text{fun } f \mapsto e_0 \\
& \mid \text{let } x = \text{ie}_1 \text{ in } e_2 \mid \text{fork } \text{ie} \\
& \mid \text{channel } \text{ie} \mid \text{send } \text{ie}_1 \text{ ie}_2 \mid \text{receive } \text{ie} \\
& \mid \text{ch } \mid \text{bind } \rho \text{ in } \text{ie} \mid \text{close } t \text{ in } \rho \\
\text{v} & \in \text{Val} \quad \text{(values)} \\
\text{v} := & c \mid \text{ch } \mid \text{close } t \text{ in } \rho \\
\rho & \in \text{Env} \quad \text{(environments)} \\
\rho := & [] \mid \rho[x \mapsto v] \\
\text{ch} & \in \text{ChId} \quad \text{(channel identifiers)}
\end{align*}
$$

The "top-level" semantics of the sequential subset of the language is given by the transition system $\rho \vdash \text{ie} \rightarrow \text{ie}'$ of Table 1. To overcome the restriction to "top-level" we shall make use of evaluation contexts $E$:

$$
E := [] \mid (E e_2) \mid (v E) \mid (\text{if } E \text{ then } e_1 \text{ else } e_2) \\
\mid (\text{let } x = E \text{ in } e_2) \mid (\text{fork } E) \mid (\text{channel } E) \\
\mid (\text{send } e_2) \mid (\text{send } v E) \mid (\text{receive } E) \\
\mid (\text{bind } \rho \text{ in } E)
$$

We have been very deliberate in when to use intermediate expressions and when to use expressions. Since we do not evaluate the body of a function before it is applied we continue to let the body be an expression rather than an intermediate expression. Similar remarks apply to the branches of the conditional and the body of the local definitions. Note that although an environment only records the terms $\text{fn } x \mapsto e_0$ and $\text{fun } f \mapsto e_0$ occurring in the closures, we do not lose the identity of the function abstractions as $e_0$ will be of the form $\text{ch}_{\rho}$ and $\lambda_0$ may be used to "uniquely" identify the function abstraction.

In order not to lose the identity of the channel identifiers we shall introduce a finitary function

$$
K : \text{ChId} \rightarrow \text{Term}
$$

that specifies which occurrence of channel $e$ that gives rise to the channel identifier. The concurrent semantics is given by the transition system $K, \text{PP} \rightarrow K', \text{PP}'$ of Table 1. Configurations have the form $K, \text{PP}$ where $K$ is as above and $\text{PP}$ is a finitary mapping from process identifiers $\rho \in \text{PId}$ to intermediate expressions; these intermediate expressions will always be closed (having no free variables) but they may contain channel identifiers that are present in the domain of $K$.

In order to evaluate inside a bind-construct we have to reconstruct the local environment. For this we use the function $\mathcal{E}$ defined by:

$$
\begin{align*}
\mathcal{E}(\rho, []) & = \rho \\
\mathcal{E}(\rho, (E e_2)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (v E)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{if } E \text{ then } e_1 \text{ else } e_2)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{let } x = E \text{ in } e_2)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{fork } E)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{channel } E)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{send } e_2)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{send } v E)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{receive } E)) & = \mathcal{E}(\rho, E) \\
\mathcal{E}(\rho, (\text{bind } \rho \text{ in } E)) & = \mathcal{E}(\rho, E)
\end{align*}
$$

**Remark** One might have expected the rule for the fork-construct to look like:

$$
K, \text{PP} : e \rightarrow K, \text{PP} : e'[\{2 \rightarrow \{2'}\]}
$$

if $p_2 \text{ dom}(\text{PP}) \cup \{p_2\}$

However, with this approach it is unclear what labels to put instead of the questions marks.

3 Abstract Closure Analysis

The result of the 0-CFA analysis for CML is a triple $(\widehat{\mathcal{C}}, \widehat{\rho}, \widehat{\kappa})$

- $\widehat{\mathcal{C}}$ is the abstract cache associating an abstract value with each labelled program point; this denotes the set of values that the program point could evaluate to;
- $\widehat{\rho}$ is the abstract environment associating an abstract value with each variable; this denotes the set of values that the variable could be bound to; and
- $\widehat{\kappa}$ is the abstract channel environment which associates an abstract value with each labelled program point denoting a channel; this denotes the set of values that could be transmitted over a channel created at that program point.

This is made precise by the following definitions:

$$
\begin{align*}
\widehat{v} & \in \widehat{\text{Val}} = \text{P}(\text{Term}) \\
\widehat{c} & \in \widehat{\text{Cache}} = \text{Lab} \rightarrow \widehat{\text{Val}} \\
\widehat{\rho} & \in \widehat{\text{Env}} = \text{Var} \rightarrow \widehat{\text{Val}} \\
\widehat{\kappa} & \in \widehat{\text{KEnv}} = \text{Lab} \rightarrow \widehat{\text{Val}}
\end{align*}
$$

Here an abstract value $\widehat{v}$ only records a set of terms of form $\text{fn } x \mapsto e, \text{fun } f \mapsto e, \text{or channel } e'$. It does not record abstract versions of the corresponding local environments as these are collapsed into the global environment $\widehat{\rho}$ in 0-CFA analyses; also it means that the analysis will be unable to record any form of causality. So for functions we record the corresponding abstraction in the program as usual and for channels we record the occurrence of channel $e'$ that gives
rise to the channel. However, \( t \) is not stable under evaluation and therefore we decide to use its final value \( Q \) instead. We will not be recording any constants among the abstract values and we thus obtain a “pure” control flow analysis with no data flow analysis component. As previously mentioned it is straightforward to extend the development to record simple properties of simple data structures (like detection of signs for integers); algebraic data structures requires a little more work as does the use of infinite sets of properties. We do not need to assume that all bound variables are distinct but clearly greater precision is achieved if this is the case; parts of the development will need to assume that all labels are distinct (see Section 5).

We shall shortly formulate the correctness of the analysis as a subject reduction property and this means that we will need to analyse a pool \( PP \) of processes. Each process identifier \( p \) is mapped to an intermediate expression so we will also have to specify the analysis for the intermediate terms \( \text{ch} \), close \( t \) in \( \rho \), and bind \( \rho \) in \( e \). The general formulation of the 0-CFA analysis will therefore have the form

\[ (\overline{\text{C}}, \overline{\rho}, \overline{\kappa}) \models_K \text{ie} \]

and it expresses that \((\overline{\text{C}}, \overline{\rho}, \overline{\kappa})\) is an acceptable closure analysis of the intermediate expression \( \text{ie} \) where the channel identifiers are mapped to their syntactic creation points by the function \( K \). The analysis of expressions is specified in Table 2 and it is extended to an analysis of a process pool by

\[ (\overline{\text{C}}, \overline{\rho}, \overline{\kappa}) \models_K \text{PP} \text{ iff } \forall p \in \text{dom}(\text{PP}) : (\overline{\text{C}}, \overline{\rho}, \overline{\kappa}) \models_K \text{PP}(p) \]

The clauses of Table 2 contain a number of inclusions of the form

\[ \text{lhs} \subseteq \text{rhs} \]

where \( \text{rhs} \) is of the form \( \overline{\text{C}}(l), \overline{\rho}(x) \) or \( \overline{\kappa}(l) \) and where \( \text{lhs} \) is of the form \( \overline{\text{C}}(l), \overline{\rho}(x), \overline{\kappa}(l) \), or \( \{t\} \). These inclusions express how the higher-order entities may flow through the program: actual parameters flow into formal parameters, results of function calls flow back to the call sites, etc.

A number of clauses also contain “recursive calls” to the acceptability relation \((\models_K)\). The noteworthy exceptions to this pattern are the clauses (fn), (fun), and (close) that in-

| (seq) | \( K, PP[p : E[c_1]] \rightarrow K, PP[p : E[c_2]] \) if \( E(\emptyset, E) \models e_1 \rightarrow e_2 \) |
| (chan) | \( K, PP[p : E[(\text{channel } O^1)] \rightarrow K[ch \rightarrow \text{channel } O^1], PP[p : E[ch]]\) if \( ch \notin \text{dom}(K) \) |
| (fork_in) | \( K, PP[p : E[(\text{fork } close \text{ (fun } x \Rightarrow t^2) \in \rho^1)] \rightarrow K, PP[p' : E[O^1]] \) if \( p' \notin \text{dom}(PP) \cup \{p\} \) |
| (fork_out) | \( K, PP[p : E[(\text{fork } close \text{ (fun } f \Rightarrow t^2) \in \rho^1)] \rightarrow K, PP[p' : E[O^1]] \) if \( p' \notin \text{dom}(PP) \cup \{p\} \) |
| (comm) | \( K, PP[p_1 : E_1[(\text{send } ch^1 \rightarrow t^2^1)] \rightarrow K, PP[p_1 : E_1[v^1]] \) if \( p_1 \neq p_2 \) |

Table 1: Small-Step Semantics
stead rely on \((\text{app})\) and \((\text{fork})\) to perform the “recursive calls” on all closures that are possibly applied. As a consequence some expressions will be analysed more than once since they may be applied at different program points whereas other will not be analysed at all since they will not be reachable. The latter is a phenomenon common in program analysis, where there is no need to analyse unreachable fragments, but is different from the perspective of type inference, where even unreachable fragments must be correctly typed.

To explain the clauses \((\text{close})\) and \((\text{bind})\) we shall first introduce the correctness relation \(\mathcal{R}_K\) that expresses when an environment of the semantics is correctly modelled by an abstract environment of the analysis:

\[
\rho \mathcal{R}_K \hat{\rho}\quad \text{if}\quad \forall x \in \text{dom}(\rho) \subseteq \text{dom}(\hat{\rho}): \\
\rho(x) \forall \hat{\rho}(\hat{\rho}(x))
\]

\[
v \mathcal{V}_K (\hat{\rho}, \hat{\nu})\quad \text{if}\quad (\forall \hat{\hat{\nu}}. v = (\text{close} t \text{ in } \rho) \Rightarrow (t \in \hat{\nu}) \wedge (\rho \mathcal{R}_K \hat{\rho})) \\
\vee (\forall \hat{ch}. (v = \text{ch}) \Rightarrow (K(\text{ch}) \in \hat{\nu}))
\]

| (const) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} c \text{' if } \text{true} \) |
| (var) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} \ x' \text{' if } \tilde{\rho}(x) \subseteq \tilde{C}(l) \) |
| (fun) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} \ (\text{fn } x \Rightarrow e)' \text{' if } (\text{fn } x \Rightarrow e) \in \tilde{C}(l) \) |
| (app) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{it} \ i_1') \text{' if } \text{true} \) |
| (if) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{if } i_e' \text{' else } i_e') \text{' if } (\text{if } i_e' \text{' else } i_e') \in \tilde{C}(l) \) |
| (let) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{let } x = i_1' \text{' in } i_2') \text{' if } (\text{let } x = i_1' \text{' in } i_2') \in \tilde{C}(l) \) |
| (fork) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{fork } i_1') \text{' if } (\text{fork } i_1') \in \tilde{C}(l) \) |
| (send) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{send } i_1') \text{' if } (\text{send } i_1') \in \tilde{C}(l) \) |
| (receive) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{receive } i_1') \text{' if } (\text{receive } i_1') \in \tilde{C}(l) \) |
| (ch) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} \text{ch'} \text{' if } K(\text{ch}) \in \tilde{C}(l) \) |
| (close) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{close } t \text{ in } \rho') \text{' if } t \in \tilde{C}(l) \wedge \rho \mathcal{R}_K \tilde{\rho} \) |
| (bind) | \((\tilde{C}, \tilde{\rho}, \tilde{\kappa}) \mathcal{K} (\text{bind } \rho \text{ in } i_1') \text{' if } (\text{bind } \rho \text{ in } i_1') \subseteq \tilde{C}(l) \wedge \rho \mathcal{R}_K \tilde{\rho} \) |

Table 2: Abstract Closure Analysis \((\mathcal{K})\)
These relations are defined mutually recursively in terms of one another. Note that in the definition of V ושל the local environment $\rho$ is related to the global abstract environment $\hat{\rho}$ as the abstract value $\hat{v}$ does not contain an abstract environment. Also note that the semantic entities (values $v$ and environments $\rho$) decrease in size as we perform the “recursive calls”; thus a simple well-founded induction in the finite size of the semantic entities suffices for showing well-definedness of these relations.

Finally, we need to clarify that the clauses of Table 2 indeed define a relation. The difficulty here is that the clauses for (app) and (fork) are not of a form that allows to define $(\hat{C}, \hat{\rho}, \hat{\kappa}) \models K$ ie by structural induction in the intermediate expression $ie$. Instead we note [13] that all right hand sides are monotone in $\hat{v}$ and hence we can define $(\hat{C}, \hat{\rho}, \hat{\kappa}) \models K ie$ as the greatest fixed point of the above specification; this is often called a coinductive definition.

Properties of the Analysis

We shall formulate semantic correctness of the analysis as a subject reduction result:

**Theorem 1 Semantic Correctness**

If $K, PP \rightarrow K', PP'$ and $(\hat{C}, \hat{\rho}, \hat{\kappa}) \models K PP$ then $(\hat{C}, \hat{\rho}, \hat{\kappa}) \models K PP'$.

Thus the analysis results obtained for the initial configuration will also hold for all subsequent configurations.

Having defined the analysis in Table 2 it is natural to ask for each program, whether or not it admits a closure analysis for each program, whether or not it admits a closure analysis of the analysis of Table 2 to track the reactivity of subexpressions explicitly (Section 4), then we modify it such that subexpressions are analysed at most once (Section 5), and finally we introduce the syntax-directed procedure for collecting constraints (Section 6).

We shall extend the analysis of Table 2 to track the reactivity of the subexpressions explicitly. To this end we define

$$\hat{R} \in \text{Reach} \Rightarrow \text{Lab} \rightarrow \mathcal{P}(\{\text{em}\})$$

The idea is that if $\hat{R}(l) = 0$ then the program point $l$ is not reachable and hence the corresponding expression will not be analysed. For this to work (in Section 5) we shall demand that all expressions in $PP_s$ are uniquely labelled (but we still do not demand that variables are unique).

Note that the claim $(\hat{C}, \hat{\rho}, \hat{\kappa}) \subseteq (\hat{C}^2, \hat{\rho}^2, \hat{\kappa}^2)$ is for all practical purposes equivalent to the claim $(\hat{C}, \hat{\rho}, \hat{\kappa}) \subseteq \text{Cache} \times \text{Env} \times K\text{Env}$ where $\text{Cache} = \text{Lab} \rightarrow \text{Val}$, $\text{Env} = \text{Var} \rightarrow \text{Val}$, $K\text{Env} = \text{Lab} \rightarrow \text{Val}$ and $Val = \mathcal{P}(\text{Term})$. Thus the condition $(\hat{C}, \hat{\rho}, \hat{\kappa}) \subseteq (\hat{C}^2, \hat{\rho}^2, \hat{\kappa}^2)$ expresses that only terms “occurring” in $PP_s$ are present in the range of $\hat{C}, \hat{\rho}$ and $\hat{\kappa}$.

**Theorem 2 Moore Family Property**

Both of the sets $\{(\hat{C}, \hat{\rho}, \hat{\kappa}) \mid (\hat{C}, \hat{\rho}, \hat{\kappa}) \models K PP\}$ and $\{(\hat{C}, \hat{\rho}, \hat{\kappa}) \subseteq (\hat{C}^2, \hat{\rho}^2, \hat{\kappa}^2) \mid (\hat{C}, \hat{\rho}, \hat{\kappa}) \models K PP\}$ are Moore families and are independent of $K$; their greatest elements are $(\text{Term}, \lambda x. \text{Term}, \lambda \text{Term})$ and $(\hat{C}^2, \hat{\rho}^2, \hat{\kappa}^2)$ respectively; also they have the same least element.

This result shows that for the purpose of finding (least) solutions it suffices to restrict $(\hat{C}, \hat{\rho}, \hat{\kappa}) \subseteq \text{Cache} \times \text{Env} \times K\text{Env}$ to $(\hat{C}, \hat{\rho}, \hat{\kappa}) \subseteq \text{Cache} \times \text{Env} \times K\text{Env}$. To remind us of this we shall allow to write $(\hat{C}, \hat{\rho}, \hat{\kappa}) \models K$ and from now on we shall assume that $(\hat{C}, \hat{\rho}, \hat{\kappa})$ is restricted in this manner.

One can prove that the Moore family property would fail to hold for an inductive rather than coinductive definition of $PP_s$.
(const) \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t e\) iff

(var) \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t x^i\) iff \((\text{on} \in \hat{R}(l) \Rightarrow \hat{\rho}(x) \subseteq \hat{C}(l))\)

(fin) \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t (\text{fn } x \Rightarrow e)^i\) iff \((\text{on} \in \hat{R}(l) \Rightarrow (\text{fn } x \Rightarrow e) \in \hat{C}(l))\)

(fun) \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t (\text{fun } f \ x \Rightarrow e)^i\) iff \((\text{on} \in \hat{R}(l) \Rightarrow (\text{fun } f \ x \Rightarrow e) \in \hat{C}(l))\)

(app) \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t (t^i_1 \ t^i_2)\)

if \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t t^i_1 \wedge (\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^t t^i_2 \wedge (\forall (\text{fn } x \Rightarrow 0^i_0) \in \hat{C}(l_1) : (R, C, \hat{\rho}, k) \models^t R^0_0 \wedge (\text{on} \in \hat{R}(l) \Rightarrow \text{on} \in \hat{R}(l_0)) \wedge (\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(l_2) \subseteq R(x)) \wedge (\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(0) \subseteq \hat{C}(l)) \wedge (\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(l_2) \subseteq \hat{C}(l))\)

if \((\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(l_2) \subseteq \hat{C}(l))\)

if \((\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(l_2) \subseteq \hat{C}(l))\)

if \((\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(l_2) \subseteq \hat{C}(l))\)

if \((\text{on} \in \hat{R}(l) \Rightarrow \hat{C}(l_2) \subseteq \hat{C}(l))\)
The analysis is extended to process pools as follows:

\[(\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP \text{ if and only if} \forall p \in dom(PP) : \forall I : PP(p) = \ell' \Rightarrow (\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} \ell' \wedge \ell \in \hat{R}(I)\]

It is useful to define

\[R_{p}^{*}(I) = \begin{cases} 
0 & \text{if } I \notin Lab, \\
\{\ell\} & \text{if } I \in Lab,
\end{cases}\]

as we then have

**Theorem 3** Preservation of Solutions

If \((\hat{C}, \hat{\rho}, \hat{k}) \models_{K} PP\), then \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP\). If \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP\), then \((\hat{C}, \hat{\rho}, \hat{k}) \models_{K} PP\). Here PP is as above and K is arbitrary.

In analogy with Theorem 2 we get that also

\[
\{(\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) | (\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP\}
\]

is a Moore family so in particular the least analysis result obtained using Table 3 will correspond to the least analysis result obtained using Table 2 and vice versa.

5 Syntax-Directed Specification

We shall now reformulate the specification of the closure analysis in Table 3 such that all subexpressions are analysed at most once. So the aim will be to use the explicit reachability information to guide when the body of function abstractions have to be analysed and then to proceed in a syntax-directed manner. The judgements have the form

\[(\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} e\]

much as in the previous section and the clauses are given in Table 4. Note that the the two function abstractions give rise to "recursive calls" of the analysis and that the clauses for function application and process creation only contain "recursive calls" for proper subexpressions. Consequently the recursive definition of \[\models_{P}\] has precisely one solution.

The component \[\hat{R}\] controls whether or not to impose the appropriate inclusions between the flow information. The analysis is extended to process pools as follows:

\[(\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP \text{ if} \forall p \in dom(P) : \forall I : PP(p) = \ell' \Rightarrow (\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} \ell' \wedge \ell \in \hat{R}(I)\]

The following result says that the analysis of Table 4 admits the same results as that of Table 3:

**Theorem 4** Preservation of Solutions

Suppose that all labels of PP are unique. If \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} PP\), then \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP\). If \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k})\) is such that \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP\), then \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models PP\).

Technically, this is the hardest of the theorems to prove, because Tables 3 and 4 have different perspectives upon what is "global" and what is "local"; see the Appendix for details. Intuitively, it is the leastness of the \[\hat{R}\] component that is essential.

6 Constraint Formulation

We are now ready to consider efficient ways of finding the least solution \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k})\) such that \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} PP\). To do so we first construct a finite set \(C_{P}\) of constraints of the forms

\[
\{u \subseteq rsh\} \quad (1)
\]

\[
\{u_{1} \subseteq rsh_{1} \Rightarrow rhs \subseteq rsh_{2}\} \quad (2)
\]

\[
\{u_{1} \subseteq rsh_{1} \Rightarrow u_{2} \subseteq rsh_{2}\} \quad (3)
\]

where \(rsh\) is of the form \(\hat{R}(I), \hat{C}(I), \hat{\rho}(x)\) or \(\hat{k}(I)\); \(lhs\) is of the form \(\hat{C}(I), \hat{\rho}(x), \hat{k}(I)\), or \(\{u\}\); and \(u\) is of the form on or \(t\). All occurrences of \(t\) are of the form \(\mathbf{fn} \ x \Rightarrow e\) or channel \(O\). The constraints are obtained by expanding \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} PP\) into a finite set of constraints of the above form and then let \(C_{P}\) be the set of individual conjuncts; at the same time we change all occurrences of \("\Rightarrow\"\) into \("\Leftarrow\"\) to avoid confusion: so \(\hat{C}(I)\) will be a set of terms whereas \(\hat{C}(I)\) is a formal symbol and similarly for \(\hat{\rho}(x)\) and \(\hat{\rho}(x)\).

To make use of the set \(\{\alpha\}\), of subterms in order to generate only a finite number of constraints.

If the size of \(PP\) is \(n\) then it might seem that there could be \(O(n)\) constraints of form (1), \(O(n)\) constraints of form (2) and \(O(n)\) constraints of form (3). However, inspection of the definitions of \(C_{P}\) shows that there will be at most \(O(n)\) constraints of form (1) and \(O(n)\) constraints of form (3).

It is important to stress that while \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models_{P} PP\) is a logical formula, \(C_{P}\) is a set of formal symbols. We can turn the latter into a logical formula first translating the \("\Leftarrow\"\) symbols into the sets \(\{\alpha\}\):

\[
\{\{\{t\}\} \Rightarrow \{t\}\} \quad (1)
\]

\[
\{\{\{\ell\}\} \Rightarrow \{\ell\}\} \quad (2)
\]

\[
\{\{\{\ell\}\} \Rightarrow \{\ell\}\} \quad (3)
\]

where \(C_{P}\) is given in Table 5. It makes use of the set \(\{\alpha\}\), of subterms in order to generate only a finite number of constraints.
We can then show that Tables 4 and 5 have the same solutions:

**Theorem 5** *Preservation of Solutions*

\((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^\to PP.* if and only if \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^\var C, [PP.]*.

It follows that the least solution \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k})\) to \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^\to PP.* equals the least solution to \((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^C C, [PP.]*.

Solving the Constraints

The constraints generated by \(C, [PP.]* can be solved using standard techniques. One possibility is to use a graph formulation of the constraints. The graph will have nodes \(R(l),\)

<table>
<thead>
<tr>
<th>Table 4: Syntax-Directed Analysis ((\models^\to))</th>
</tr>
</thead>
</table>

This definition can be lifted to sets of constraints by:

\((\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^\to C \iff \forall C \in C : (\hat{R}, \hat{C}, \hat{\rho}, \hat{k}) \models^\to C\)
The graph will have edges for a subset of the constraints in $\mathcal{C}_l$. Thus the resulting graph has order $\mathcal{O}(n)$ nodes and at most $\mathcal{O}(n^2)$ edges.

Thus the resulting graph has $\mathcal{O}(n)$ nodes and at most $\mathcal{O}(n^2)$ edges.

Having constructed the graph we now traverse all edges in order to propagate information from one $\mathcal{D}[q]$ to another $\mathcal{D}[q']$. We make certain only to traverse an edge from $q_1$ to $q_2$ when $\mathcal{D}[q_1]$ is extended with a term not previously there. Furthermore, an edge decorated with a constraint $\{u\} \subseteq q \Rightarrow q_1 \subseteq q_2$ is only traversed if in fact $u \in \mathcal{D}[q_1]$ and similarly an edge decorated with a constraint $\{u\} \subseteq q \Rightarrow \{u'\} \subseteq q'$ is only traversed if $u \in \mathcal{D}[q]$. For a node $q$ the data field $\mathcal{D}[q]$ is a value in $\mathcal{P}(\mathcal{O}(n^2))$ so it can be increased at most $\mathcal{O}(n)$ times. Since we make sure only to traverse an edge when a new term can be added this gives an overall bound of $\mathcal{O}(n^3)$ for solving the constraints, which is the best result known for 0-CFA analysis. Further details of the algorithm may be found in the Appendix.

### 7 Conclusion

We have shown how the core methodology of abstract interpretation can be adapted to the specification of control flow analyses. The starting point has been an abstract specification of a 0-CFA analysis for CML that is proved seman-
Theorem 1 Semantic Correctness  
If $K, PP \rightarrow K', PP'$ and $(\hat{C}, \hat{p}, \hat{\kappa}) \models K PP$ then $(\hat{C}, \hat{p}, \hat{\kappa}) \models K PP'$.

Proof The proof is by case analysis on the semantics: for the case [seq] we use Proposition 9 and for the remaining cases we use Lemmas 6 and 7.

Proof of Theorem 2

Proposition 10 Both of the sets $\{\hat{C}, \hat{p}, \hat{\kappa} \mid (\hat{C}, \hat{p}, \hat{\kappa}) \models K PP\}$ and $\{\hat{C}, \hat{p}, \hat{\kappa} \subseteq (\hat{C}, \hat{p}, \hat{\kappa}) \mid (\hat{C}, \hat{p}, \hat{\kappa}) \models K PP\}$ are Moore families for all $e \in Exp$, and are independent of $K$: their greatest elements are $(M, Term, \lambda x, \lambda y, Term, M)$ and $(\hat{C}, \hat{p}, \hat{\kappa})$, respectively; also they have the same least element.

Proof A straightforward application of coinduction.

Theorem 2 Moore Family Property

Both of the sets $\{\hat{C}, \hat{p}, \hat{\kappa} \mid \{\hat{C}, \hat{p}, \hat{\kappa} \} \models K PP\}$ and $\{\hat{C}, \hat{p}, \hat{\kappa} \subseteq \{\hat{C}, \hat{p}, \hat{\kappa}\} \mid \{\hat{C}, \hat{p}, \hat{\kappa}\} \models K PP\}$ are Moore families and are independent of $K$: their greatest elements are $(M, Term, \lambda x, \lambda y, Term, M)$ and $(\hat{C}, \hat{p}, \hat{\kappa})$, respectively; also they have the same least element.

Proof Immediate by Proposition 10.

Proof of Theorem 3

Proposition 11 For all expressions $e \in Exp$, all labels $l \in Lab$, if $(\hat{C}, \hat{p}, \hat{\kappa}) \models K e$ then $(\hat{R}, \hat{C}, \hat{p}, \hat{\kappa}) \models e$; and if $(\hat{R}, \hat{C}, \hat{p}, \hat{\kappa}) \models e$ and $\alpha \in R(l)$ then $(\hat{C}, \hat{p}, \hat{\kappa}) \models \alpha$.

Proof For the first result we proceed as follows. First write

$$(\hat{C}, \hat{p}, \hat{\kappa}, K, l, e) \in \models_e$$

for $(\hat{C}, \hat{p}, \hat{\kappa}) \models K e$ and

$$(\hat{C}, \hat{p}, \hat{\kappa}, K, l, e) \in \models_e$$

where the relation $\models_e$ is defined in Table 2 and the relation $\models_e$ is defined in Table 3. Then note that these tables define monotonic functions $F_2$ and $F_3$ such that $\models_2 = gF_2$ and $\models_3 = gF_3$. Next we prove

$$F_3(\models_2) \subseteq F_3(\models_3)$$

by inspection of all clauses. Finally $\models_1 \subseteq \models_3$ follows by coinduction.

For the second result we write

$$(\hat{R}, \hat{C}, \hat{p}, \hat{\kappa}, K, l, e) \in \models_e$$

for $(\hat{C}, \hat{p}, \hat{\kappa}) \models K e$ and

$$(\hat{R}, \hat{C}, \hat{p}, \hat{\kappa}, K, l, e) \in \models_e$$

where again $\models_2 = gF_2$ and $\models_3 = gF_3$ for monotonic functions $F_2$ and $F_3$ given by Tables 2 and 3. Next we prove

$$F_3(\models_2) \cap ON \subseteq F_3(\models_3) \cap ON$$

by inspection of all clauses. This shows $\models_2 \cap ON \subseteq F_3(\models_3)$ and hence $\models_3 \cap ON \subseteq \models_3$ follows by coinduction.

Theorem 3 Preservation of Solutions

If $(\hat{C}, \hat{p}, \hat{\kappa}) \models K PP$, then $(\hat{R}, \hat{C}, \hat{p}, \hat{\kappa}) \models PP$. If $(\hat{R}, \hat{C}, \hat{p}, \hat{\kappa}) \models PP$, then $(\hat{C}, \hat{p}, \hat{\kappa}) \models K PP$. Here PP is as above and $K$ is arbitrary.

Proof Immediate by Proposition 11.
Lemma 12 If \(\widehat{R}, \widehat{C}, \widehat{\rho}, \widehat{K}) \models_{\varphi} PP\), then (i) \((\text{fn } x \Rightarrow \in \in Term_{\varphi}) \in Term_{\varphi}\), ensures \((\widehat{R}, \widehat{C}, \widehat{\rho}, \widehat{K}) \models_{\varphi} \in \in \in\) and (ii) \((\text{fun } x \Rightarrow \in \in \in \in ) \in Term_{\varphi}\), ensures \((\widehat{R}, \widehat{C}, \widehat{\rho}, \widehat{K}) \models_{\varphi} \in \in \in \in \in\) and 
\(\text{on } \in \widehat{R}(l_0) \Rightarrow (\text{fun } x \Rightarrow \in \in \in \in ) \in \widehat{\rho}(f)\).

Proof The result is immediate from the syntax-directed nature of Table 4 and the construction of Term_{\varphi}.

Lemma 13 Suppose that all labels of PP_{\varphi} are unique. If \((\widehat{R}, \widehat{C}, \widehat{\rho}, \widehat{K})\) is least such that \((\widehat{R}, \widehat{C}, \widehat{\rho}, \widehat{K}) \models_{\varphi} PP_{\varphi}\), then (i) \((\text{fn } x \Rightarrow \in \in \in \in ) \in Term_{\varphi}\), ensures \((\widehat{R}, \widehat{C}, \widehat{\rho}, \widehat{K}) \models_{\varphi} \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \in \i
INPUT: \( \mathcal{C}_s[PP_s] \)

OUTPUT: \( (\hat{R}, \hat{C}, \hat{\rho}, \hat{\kappa}) \)

METHOD: Step 1: Initialisation

\[
W := \text{nil};
\]
for \( q \) in Nodes do \( D[q] := \emptyset; \)
for \( q \) in Nodes do \( E[q] := \text{nil}; \)

Step 2: Building the graph

for \( c \) in \( \mathcal{C}_s[PP_s] \) do

\[
\text{case } c \text{ of}
\]
\[
\{u\} \subseteq q: \text{add}(q, \{u\});
\]
\[
q_1 \subseteq q_2: \text{if } q_1 \neq q_2 \text{ then add}(q_2, D[q_1]);
\]
\[
\{u_1\} \subseteq q_1 \Rightarrow \ldots \Rightarrow \{u_k\} \subseteq q_k \Rightarrow q_1 \subseteq q_k:
\]
\[
E[q_k] := \text{cons}(c, E[q_k]);
\]

Step 3: Iteration

while \( W \neq \text{nil} \) do

\[
q := \text{head}(W); W := \text{tail}(W);
\]
for \( c \) in \( E[q] \) do

\[
\text{case } c \text{ of}
\]
\[
q_1 \subseteq q_2: \text{add}(q_2, D[q_1]);
\]
\[
\{u_1\} \subseteq q_1 \Rightarrow \ldots \Rightarrow \{u_k\} \subseteq q_k \Rightarrow q_1 \subseteq q_k;
\]
\[
\text{if } u_1 \in D[q'] \land \ldots \land u_k \in D[q_k] \text{ then add}(q_2, D[q_1]);
\]

Step 4: Recording the solution

for \( l \) in Lab., do \( \hat{R}(l) := D[\hat{R}(l)]; \)
for \( l \) in Lab., do \( \hat{C}(l) := D[\hat{C}(l)]; \)
for \( x \) in Var., do \( \hat{\rho}(x) := D[\hat{\rho}(x)]; \)
for \( l \) in Lab., do \( \hat{\kappa}(l) := D[\hat{\kappa}(l)]; \)

USING: 

procedure \( \text{add}(q,d) \) is

\[
\text{if } \neg (d \subseteq D[q])
\]
\[
\text{then } D[q] := D[q] \cup d;
\]
\[
W := \text{cons}(q,W);
\]

Table 6: Algorithm for Solving Constraints.

Constraint Solving

For completeness we present an abstract algorithm for solving constraints. It operates on the main data structures:

- a worklist \( W \) that is a list of nodes whose outgoing edges should be traversed;
- an array \( D \) that for each node gives an element of \( \text{Var}_s \); and
- an array \( E \) that for each node gives a list of the successor nodes.

The set Nodes contains \( \hat{R}(l) \), \( \hat{C}(l) \), and \( \hat{\kappa}(l) \) for all \( l \) in Lab., and \( \hat{\rho}(x) \) for all \( x \) in Var.

The first step of the algorithm is to suitable initialise the data structures. The second step is to build the graph and to perform the initial assignments to the data fields. This is established using the procedure \( \text{add}(p,d) \) that incorporates \( d \) into \( D[p] \) and adds \( p \) to the work list if \( d \) was not part of \( D[p] \).

The third step is to continue propagating contributions along edges as long as the work list is non-empty. The fourth and final step is to record the solution in a more familiar form.

The following result shows that the algorithm does indeed compute the solution we want:

**Proposition 15**

Given input \( \mathcal{C}_s[PP_s] \) the algorithm of Table 6 terminates and the result \( (\hat{R}, \hat{C}, \hat{\rho}, \hat{\kappa}) \) produced by the algorithm satisfies

\[
(\hat{R}, \hat{C}, \hat{\rho}, \hat{\kappa}) = \bigcap \{ (\hat{R}_1, \hat{C}_1, \hat{\rho}_1, \hat{\kappa}_1) \mid (\hat{R}_1, \hat{C}_1, \hat{\rho}_1, \hat{\kappa}_1) \models \mathcal{C}_s[PP_s] \}
\]

and hence is the least solution to \( \mathcal{C}_s[PP_s] \). \( \square \)

**Proof**

It is immediate that Steps 1, 2 and 4 terminate, and this leaves us with Step 3. It is immediate that the values of \( D[q] \) never decrease and that they can be increased at most a finite number of times. It is also immediate that a node is added to the work list only if some value of \( D[q] \) actually increased. To each node placed on the work list only a finite amount of calculation (bounded by the number of outgoing edges) needs to be performed in order to remove the node from the work list. This guarantees termination.

Next let \( (\hat{R}_1, \hat{C}_1, \hat{\rho}_1, \hat{\kappa}_1) \) be a solution to \( (\hat{R}_1, \hat{C}_1, \hat{\rho}_1, \hat{\kappa}_1) \models \mathcal{C}_s[PP_s] \). It is possible to show that the following invariant
is maintained at all points after Step 1. It follows that $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) \subseteq (\hat{R}_1, \hat{C}_1, \hat{p}_1, \hat{k}_1)$ upon completion of the algorithm.

We prove that $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) \models \varphi_{\SEM{PP}}$ by contradiction. So suppose there exists $c \in \varphi_{\SEM{PP}}$ such that $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) \not\models \varphi_{\SEM{PP}}$ does not hold. If $c \models \{u\} \subseteq q$ then Step 2 ensures that $\{u\} \subseteq \mathbb{D}[q]$ and this is maintained throughout the algorithm; hence $c$ cannot have this form. If $c \models q_1 \subseteq q_2$ it must be the case that the final value of $D$ satisfies $D[q_1] \neq \emptyset$ otherwise $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) \models \varphi_{\SEM{PP}}$ would hold; now consider the last time $\mathbb{D}[q_1]$ was modified and note that $q_1$ was placed on the work list at that time (by the procedure add); since the final work list is empty we must have considered the constraint $c$ (which is in $\mathbb{E}[q_1]$) and updated $\mathbb{D}[q_2]$ accordingly; hence $c$ cannot have this form. If $c \models \{u\} \subseteq q_1 \Rightarrow \cdots \Rightarrow q_4 \Rightarrow q_1 \subseteq q_2$ it must be the case that the final value of $D$ satisfies $D[q_1] \neq \emptyset, \cdots, D[q'_i] \neq \emptyset$; now consider the last time one of $\mathbb{D}[q'_i], \cdots, D[q'_i]$ and $\mathbb{D}[q_1]$ was modified and note that $q'_i, \cdots, q'_i$, or $q_1$ was placed on the work list at that time, since the final work list is empty we must have considered the constraint $c$ and updated $\mathbb{D}[q_2]$ accordingly; hence $c$ cannot have this form.

We have now shown that $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) \models \varphi_{\SEM{PP}}$ and that $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) \subseteq (\hat{R}_1, \hat{C}_1, \hat{p}_1, \hat{k}_1)$ whenever $(\hat{R}_1, \hat{C}_1, \hat{p}_1, \hat{k}_1) \models \varphi_{\SEM{PP}}$. It now follows that $(\hat{R}, \hat{C}, \hat{p}, \hat{k}) = \cap\{(\hat{R}_1, \hat{C}_1, \hat{p}_1, \hat{k}_1) \mid (\hat{R}_1, \hat{C}_1, \hat{p}_1, \hat{k}_1) \models \varphi_{\SEM{PP}}\}$ as required.

References


