Flow Logic: A Multi-paradigmatic Approach to Static Analysis

Hanne Riis Nielson and Flemming Nielson

Informatics and Mathematical Modelling
Technical University of Denmark
{riis,nielson}@@imm.dtu.dk

Abstract. Flow logic is an approach to static analysis that separates the *specification* of when an analysis estimate is acceptable for a program from the actual *computation* of the analysis information. It allows one not only to combine a variety of programming paradigms but also to link up with state-of-the-art developments in classical approaches to static analysis, in particular data flow analysis, constraint-based analysis and abstract interpretation. This paper gives a tutorial on flow logic and explains the underlying methodology; the multi-paradigmatic approach is illustrated by a number of examples including functional, imperative, object-oriented and concurrent constructs.

1 Introduction

Computer systems often combine different programming paradigms and to support the software development phase it is desirable to have techniques and tools available for reasoning about and validating the behaviour of multi-paradigmatic programs. One such set of techniques and tools are based on static analysis. Traditionally, static analysis has been developed and used in the construction of optimising compilers but recent developments show promise of a much wider application area that includes validating that programs adhere to specific protocols and that certain performance guarantees can be ensured.

Flow logic is a formalism for static analysis that is based on logical systems. It focusses on specifying what it means for an analysis estimate to be acceptable for a program. The specification can be given at different levels of abstraction depending on the interest in implementation details but there will always be a clear distinction between specifying *what* is an acceptable analysis estimate and *how* to compute it. Flow logic is not a "brand new" technique for static analysis; rather it is firmly rooted upon existing technology and insights, in particular from the classical areas of data flow analysis [9], constraint-based analysis [10] and abstract interpretation [7]. In contrast to the traditional presentations of these approaches, flow logic applies to programs expressed in mixed programming paradigms and allows the existing techniques and insights to be integrated thereby giving a wider perspective on their applicability.

Flow logic and operational semantics. Flow logic shares many of its properties with structural operational semantics [23]. One reason for the success of structural operational semantics is that it is applicable to a wide range

T.Æ. Mogensen et al. (Eds.): The Essence of Computation, LNCS 2566, pp. 223–244, 2002.

of programming paradigms. Another is that within the same formalism, definitions can be given at different levels of abstraction reflecting the interest in implementation details; subsequently one definition can be refined into another by a transformation process or, alternatively, the relationship between different definitions can be established using various proof techniques.

The ability to specify the analysis at different abstraction levels has lead to the identification of *different styles of flow logic* just as there are different styles of operational semantics – as for example small-step versus big-step transitions, environments versus substitutions, and evaluation contexts versus explicit rules for evaluation in context. In Section 2, which takes the form of a short tutorial, we introduce two sets of styles that so far have emerged for flow logic: one being *abstract* versus *compositional* and the other being *succinct* versus *verbose*.

Flow logic and type systems. The overall methodology of flow logic has much in common with that of type systems [11]. As already mentioned we have a clear distinction between the judgements *specifying* the acceptability of an analysis estimate and the algorithms *computing* the information. Also, we are interested in the following properties that all have counterparts in type systems:

- *Semantic correctness*: this property ensures that the analysis estimates always "err on the safe side" and relates to the slogan "well-typed programs do not go wrong" (see Subsection 3.2).
- *Existence of best analysis estimates*: this property ensures that there is a most precise analysis estimate for all programs; the analogy is here to the existence of principal types in type systems (see Subsection 3.3).
- *Efficient implementations*: these typically take the form of constraint generation algorithms working in conjunction with constraint solving algorithms; as for type systems the correctness is established via syntactic soundness and completeness results (see Subsection 3.4).

However, there are also important differences between flow logic and type systems. One is that the clauses defining the judgements of a flow logic in principle have to be interpreted *co-inductively* rather than inductively: an analysis estimate for a program is acceptable if it does *not* violate any of the conditions imposed by the specification. It turns out that for the compositional specifications the co-inductive and inductive interpretations coincide whereas for the abstract specifications additional care needs to be taken to ensure well-definedness (see Subsection 3.1).

Another important difference between flow logic and type systems is that the former allows the exploitation of classical techniques for static analysis; we shall not go further into this in the present paper.

Flow logic for multiple paradigms. Flow logic was originally developed for a pure *functional* language (resembling the core of Standard ML) [14]; then it was further developed to handle *imperative* constructs (as in Standard ML) [17,21] and *concurrent* constructs (as in Concurrent ML) [8]. Later the approach has been used for various calculi of computation including *object-oriented* calculi (as the imperative object calculus) [15], the pi-calculus (modeling communication

in a pure form) [3], the ambient calculus (generalising the notion of mobility found in Java) [19] and the spi-calculus (modeling cryptographic primitives) [4].

We conclude in Section 4 by illustrating the flexibility of flow logic by giving a number of examples for different programming paradigms thereby opening up for the exploitation of static analysis techniques in a software development process integrating several paradigms.

2 A Tutorial on Flow Logic

A flow logic specification consists of a set of clauses defining a judgement expressing the acceptability of an analysis estimate for a program fragment. In principle, the specification has to be interpreted *co-inductively*: an analysis estimate can only be rejected if it does not fulfill the conditions expressed by the definition. Two sets of criteria have emerged for classifying flow logic specifications:

- *abstract* versus *compositional*, and
- *succinct* versus *verbose*.

A *compositional* specification is syntax-directed whereas an *abstract* specification is not. In general an abstract specification will be closer to the standard semantics and a compositional specification will be closer to an implementation. In the earlier stages of designing an analysis it may be useful to work with an abstract specification as it basically relies on an understanding of the semantics of the language and one can therefore concentrate on designing the abstract domains used by the analysis; this is particularly pertinent for programming languages allowing programs as data objects. Thinking ahead to the implementation stage one will often transform the specification into a compositional flow logic. Another reason for working with compositional specifications is that the associated proofs tend to be simpler as the coinductive and inductive interpretation of the defining clauses coincide.

A *verbose* specification reports all of the internal flow information as is normally the case for data flow analysis and constraint-based analysis; technically, this is achieved by having appropriate data structures (often called caches) holding the required analysis information. A specification that is not verbose is called *succinct* and it will often focus on the top-level parts of the analysis estimate in the manner known from type systems.

To give a feeling for these different styles we shall in this section present all four combinations. To keep things manageable we shall focus on a single programming paradigm; in Section 4 we extend the ideas to other paradigms.

The λ-calculus. The example language will be the λ-calculus with expressions $e \in \mathbf{Exp}$ given by

$$e ::= c \mid x \mid \lambda x_0.e_0 \mid e_1\, e_2$$

where $c \in \mathbf{Const}$ and $x \in \mathbf{Var}$ are unspecified sets of first-order constants and variables.

We choose an environment-based call-by-value big-step operational semantics [6]. The values $v \in \mathbf{Val}$ are either constants c or closures of the form $\langle \lambda x_0.e_0, \rho_0 \rangle$

where $\rho_0 \in \mathbf{Env} = \mathbf{Var} \rightarrow_{\mathsf{fin}} \mathbf{Val}$ is an environment mapping (a finite set of) variables to values. The semantics is formalised by a transition relation $\rho \vdash e \rightarrow v$ meaning that evaluating e in the environment ρ gives rise to the value v; it is defined by the following axioms and rules:

$$\rho \vdash c \rightarrow c$$
$$\rho \vdash x \rightarrow \rho(x)$$
$$\rho \vdash \lambda x_0.e_0 \rightarrow \langle \lambda x_0.e_0, \rho \rangle$$

$$\frac{\rho \vdash e_1 \rightarrow \langle \lambda x_0.e_0, \rho_0 \rangle \qquad \rho \vdash e_2 \rightarrow v_2 \qquad \rho_0[x_0 \mapsto v_2] \vdash e_0 \rightarrow v_0}{\rho \vdash e_1\, e_2 \rightarrow v_0}$$

Example 1. The expression $(\lambda x.x\,3)(\lambda y.\lambda z.y)$ evaluates to $\langle \lambda z.y, [y \mapsto 3] \rangle$ as shown by the derivation tree

$$\frac{[\,] \vdash \lambda x.x\,3 \rightarrow \mathsf{C}_x \qquad [\,] \vdash \lambda y.\lambda z.y \rightarrow \mathsf{C}_y \qquad \dfrac{\rho_x \vdash x \rightarrow \mathsf{C}_y \qquad \rho_x \vdash 3 \rightarrow 3 \qquad \rho_y \vdash \lambda z.y \rightarrow \mathsf{C}_z}{\rho_x \vdash x\,3 \rightarrow \mathsf{C}_z}}{[\,] \vdash (\lambda x.x\,3)(\lambda y.\lambda z.y) \rightarrow \mathsf{C}_z}$$

where we write $[\,]$ for the environment with empty domain and additionally make use of the abbreviations $\mathsf{C}_x = \langle \lambda x.x\,3, [\,] \rangle$, $\mathsf{C}_y = \langle \lambda y.\lambda z.y, [\,] \rangle$, $\rho_x = [x \mapsto \mathsf{C}_y]$, $\rho_y = [y \mapsto 3]$ and $\mathsf{C}_z = \langle \lambda z.y, \rho_y \rangle$. □

The aim of the analysis is to statically predict which values an expression may evaluate to. The analysis works with abstract representations of the semantic values. All constants are represented by the entity $\diamond$ so the analysis will record the presence of a constant but not its actual value. A closure $\langle \lambda x_0.e_0, \rho_0 \rangle$ is represented by an abstract closure $(\!|\lambda x_0.e_0|\!)$; a representation of the environment ρ_0 will be part of a global abstract environment $\hat{\rho} \in \widehat{\mathbf{Env}} = \mathbf{Var} \rightarrow \widehat{\mathbf{Val}}$. The judgements of the analysis will be specified in terms of *sets* $\hat{v} \in \widehat{\mathbf{Val}}$ of such abstract values and will be relative to an abstract environment. The abstract environment is *global* meaning that it is going to represent *all* the environments that may arise during the evaluation of the expression — thus, in general, a more precise analysis result can be obtained by renaming bounding occurrences of variables apart. More details of the analysis will be provided shortly.

Example 2. Continuing the above example, the analysis may determine that during the evaluation of the expression $(\lambda x.x\,3)(\lambda y.\lambda z.y)$ the variable x may be bound to $(\!|\lambda y.\lambda z.y|\!)$, y may be bound to $\diamond$, z will never be bound to anything (as the function $\lambda z.y$ is never applied) and, furthermore, the expression may evaluate to a value represented by the abstract closure $(\!|\lambda z.y|\!)$. □

2.1 An Abstract Succinct Specification

Our first specification is given by judgements of the form

$$\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$$

and expresses that the set $\hat{v}$ is an acceptable analysis estimate for the expression e in the context specified by the abstract environment $\hat{\rho}$. The analysis is defined by the clauses:

$$\begin{array}{rll}
\hat{\rho} \models_{\mathsf{as}} c : \hat{v} & \mathsf{iff} & \diamond \in \hat{v} \\
\hat{\rho} \models_{\mathsf{as}} x : \hat{v} & \mathsf{iff} & \hat{\rho}(x) \subseteq \hat{v} \\
\hat{\rho} \models_{\mathsf{as}} \lambda x_0.e_0 : \hat{v} & \mathsf{iff} & \{\!|\lambda x_0.e_0|\!\} \in \hat{v} \\
\hat{\rho} \models_{\mathsf{as}} e_1\, e_2 : \hat{v} & \mathsf{iff} & \hat{\rho} \models_{\mathsf{as}} e_1 : \hat{v}_1 \ \wedge\ \hat{\rho} \models_{\mathsf{as}} e_2 : \hat{v}_2 \ \wedge \\
 & & \forall \{\!|\lambda x_0.e_0|\!\} \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow\ [\hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge \hat{\rho} \models_{\mathsf{as}} e_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}]
\end{array}$$

The first clause expresses that the analysis estimate must contain the abstract value $\diamond$, the second clause requires that the analysis estimate includes all the values that x can take in the abstract environment $\hat{\rho}$ and the third clause states that the abstract closure $\{\!|\lambda x_0.e_0|\!\}$ must be included in the analysis estimate.

The clause for application expresses that in order for $\hat{v}$ to be an acceptable analysis estimate then it must be possible to find acceptable analysis estimates for the operator as well as the operand. Furthermore, whenever some abstract closure $\{\!|\lambda x_0.e_0|\!\}$ is a possible value of the operator and whenever the operand may evaluate to something at all (i.e $\hat{v}_2 \neq \emptyset$) then it must be the case that the potential actual parameters $\hat{v}_2$ are included in the possible values for the formal parameter x_0; also, there must exist an acceptable analysis estimate $\hat{v}_0$ for the body e_0 that is included in the analysis estimate $\hat{v}$ of the application itself. Note that $\hat{v}_0$, $\hat{v}_1$ and $\hat{v}_2$ only occur on the right hand side of the clause; it is left implicit that they are *existentially quantified*.

Example 3. Returning to the expression $(\lambda x.x\,3)(\lambda y.\lambda z.y)$ of the previous examples let us verify that

$$\hat{\rho} \models_{\mathsf{as}} (\lambda x.x\,3)(\lambda y.\lambda z.y) : \{\!|\lambda z.y|\!\}\}$$

is an acceptable analysis estimate in the context given by:

$$\hat{\rho} = \begin{array}{|c|c|c|}
\hline
x & y & z \\
\hline
\{\!|\lambda y.\lambda z.y|\!\}\} & \{\diamond\} & \emptyset \\
\hline
\end{array}$$

First we observe that the clause for λ-abstraction gives:

$$\hat{\rho} \models_{\mathsf{as}} \lambda x.x\,3 : \{\!|\lambda x.x\,3|\!\}\} \text{ and } \hat{\rho} \models_{\mathsf{as}} \lambda y.\lambda z.y : \{\!|\lambda y.\lambda z.y|\!\}\}$$

This establishes the first two conditions of the clause for application. To verify the third condition we only have to consider $\{\!|\lambda x.x\,3|\!\}$ and we have to *guess* an analysis estimate for the body $x\,3$. One such guess gives us the proof obligations:

$$\{\!|\lambda y.\lambda z.y|\!\}\} \subseteq \hat{\rho}(x) \text{ and } \hat{\rho} \models_{\mathsf{as}} x\,3 : \{\!|\lambda z.y|\!\}\} \text{ and } \{\!|\lambda z.y|\!\}\} \subseteq \{\!|\lambda z.y|\!\}\}$$

The first and the third of these are trivial and to verify the second condition we apply the clause for application once again; we dispense with the details.

Note that $\hat{\rho}(z) = \emptyset$. This reflects that z is never bound to anything, or equivalently, that $\lambda z.y$ is never applied. □

The above specification is *abstract* because the body of a λ-abstraction is only required to have an acceptable analysis estimate if the λ-abstraction might be applied somewhere. This follows the tradition of data flow analysis and abstract interpretation where only reachable code it analysed and it is in contrast to

the more type theoretic approaches where all subexpressions are required to be typable. Note that we may be required to find several analysis estimates for the body of a single λ-abstraction as it may be applied in many places; in the terminology of constraint-based analysis we may say that the analysis is polyvariant.

The specification is *succinct* because the occurrence of $\hat{v}$ in $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ expresses the overall analysis estimate for e; if we want details about the analysis information for subexpressions of e then we have to inspect the reasoning leading to the judgement $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$.

It is interesting to note that the specification applies to *open systems* as well as closed systems since λ-abstractions are analysed when they are applied; hence they are not required to be part of the program of interest but may for example be part of library routines.

2.2 A Compositional Succinct Specification

A compositional specification of the same analysis will analyse the body of λ-abstractions at their definition point rather than at their application point. This means that we need some way of linking information available at these points and for this we introduce a global cache $\hat{C}$ that will record the analysis estimates for the bodies of the λ-abstractions. In order to refer to these bodies we shall extend the syntax with labels $\ell \in \mathbf{Lab}$

$$e ::= c \mid x \mid \lambda x_0.e_0^{\ell_0} \mid e_1\, e_2$$

and we take $\hat{C} \in \widehat{\mathbf{Cache}} = \mathbf{Lab} \to \widehat{\mathbf{Val}}$. The labels allow us to refer to specific program points in an explicit way and they allow us to use the cache as a global data structure but they play no role in the semantics; for optimal precision the subexpressions should be uniquely labelled. The judgements of our analysis now have the form

$$(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$$

and expresses that $\hat{v}$ is an acceptable analysis estimate for e in the context specified by $\hat{C}$ and $\hat{\rho}$. The analysis is defined as follows:

$$\begin{array}{rll}
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} c : \hat{v} & \mathsf{iff} & \diamond \in \hat{v} \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} x : \hat{v} & \mathsf{iff} & \hat{\rho}(x) \subseteq \hat{v} \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} \lambda x_0.e_0^{\ell_0} : \hat{v} & \mathsf{iff} & (\!|\lambda x_0.e_0^{\ell_0}|\!) \in \hat{v} \wedge \\
 & & \hat{\rho}(x_0) \neq \emptyset \Rightarrow [(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_0^{\ell_0} : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{C}(\ell_0)] \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_1\, e_2 : \hat{v} & \mathsf{iff} & (\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_1 : \hat{v}_1 \ \wedge \ (\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e_2 : \hat{v}_2 \wedge \\
 & & \forall (\!|\lambda x_0.e_0^{\ell_0}|\!) \in \hat{v}_1 : \hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge \hat{C}(\ell_0) \subseteq \hat{v}
\end{array}$$

As before the third clause expresses that $(\!|\lambda x_0.e_0^{\ell_0}|\!)$ must be included in the analysis estimate but it additionally requires that *if* the λ-abstraction is ever applied to some value *then* the body e_0 has an acceptable analysis estimate $\hat{v}_0$ that is included in the cache for ℓ_0. The check of whether or not the λ-abstraction is applied is merely a check of whether the abstract environment specifies that some

actual parameters may be bound to x_0. The clause for application is as before with the exception that we consult the appropriate entry in the cache whenever we need to refer to an analysis estimate for the body of a λ-abstraction.

Example 4. We shall now annotate the example as $(\lambda x.(x\,3)^1)(\lambda y.(\lambda z.y^3)^2)$ and show that

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} (\lambda x.(x\,3)^1)(\lambda y.(\lambda z.y^3)^2) : \{\lfloor \lambda z.y^3 \rfloor\}$$

is an acceptable analysis estimate in the context where:

$$\hat{C} = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \{\lfloor \lambda z.y^3 \rfloor\} & \{\lfloor \lambda z.y^3 \rfloor\} & \emptyset \\ \hline \end{array} \quad \text{and} \quad \hat{\rho} = \begin{array}{|c|c|c|}\hline x & y & z \\ \hline \{\lfloor \lambda y.(\lambda z.y^3)^2 \rfloor\} & \{\diamond\} & \emptyset \\ \hline \end{array}$$

The clause for application requires that we *guess* acceptable analysis estimates for the operator and operand; one such guess leaves us with the proof obligations

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda x.(x\,3)^1 : \{\lfloor \lambda x.(x\,3)^1 \rfloor\} \text{ and } (\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda y.(\lambda z.y^3)^2 : \{\lfloor \lambda y.(\lambda z.y^3)^2 \rfloor\}$$

together with $\{\lfloor \lambda y.(\lambda z.y^3)^2 \rfloor\} \subseteq \hat{\rho}(x)$ and $\hat{C}(1) \subseteq \{\lfloor \lambda z.y^3 \rfloor\}$. Let us concentrate on the second judgement above where we will use the clause for λ-abstraction. Clearly $\lfloor \lambda y.(\lambda z.y^3)^2 \rfloor \in \{\lfloor \lambda y.(\lambda z.y^3)^2 \rfloor\}$ and since $\hat{\rho}(y) \neq \emptyset$ we have to *guess* an acceptable analysis estimate for the body of the λ-abstraction. It is here sufficient to show:

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda z.y^3 : \{\lfloor \lambda z.y^3 \rfloor\} \text{ and } \{\lfloor \lambda z.y^3 \rfloor\} \subseteq \hat{C}(2)$$

The latter is trivial and since $\hat{\rho}(z) = \emptyset$ we immediately get the former from the clause for λ-abstraction. Note that $\hat{C}(3) = \emptyset$; this reflects that the body of the function $\lambda z.y^3$ is never evaluated as the function is never called. □

Clearly the specification is *compositional*; it still has *succinct* components since the $\hat{v}$ of $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$ gives the analysis information for the overall expression e; to get hold of the analysis information of subexpressions we have to inspect the reasoning leading to the judgement $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$. In contrast to the abstract specification, we only have to find one acceptable analysis estimate for each subexpression; in the terminology of constraint-based analysis we may say that the analysis is monovariant.

Relationship between the specifications. The above specification is restricted to *closed* systems since the bodies of λ-abstractions only are analysed at their definition points. If we restrict our attention to closed systems then the above specification can be seen as a refinement of the one of Subsection 2.1 in the following sense:

> Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e : \hat{v}$ and $\hat{C}$ and $\hat{\rho}$ only mention λ-abstractions occurring in e then $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ (modulo the labelling of expressions).

This means that an implementation that is faithful to the compositional specification also will be faithful to the abstract specification. The converse result does not hold in general; this is analogous to the insight that polyvariant analyses often are more precise than monovariant ones. In [8,18] we present proof techniques for establishing the relationship between abstract and compositional specifications.

We may obtain an analysis closer in spirit to type systems by replacing the clause for λ-abstraction with

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda x_0.e_0^{\ell_0} : \hat{v} \quad \mathsf{iff} \quad \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{v} \ \wedge \ (\hat{C},\hat{\rho}) \models_{\mathsf{cs}} e_0^{\ell_0} : \hat{v}_0 \ \wedge \ \hat{v}_0 \subseteq \hat{C}(\ell_0)$$

where the body must have an acceptable analysis estimate even in the case where the λ-abstraction is never applied. This will give a less precise (but still semantically correct) analysis.

Example 5. With the above clause the proof of $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} \lambda z.y^3 : \{\{\!|\lambda z.y^3|\!\}\}$ breaks down: we have to show that $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} y^3 : \hat{v}_0$ and $\hat{v}_0 \subseteq \hat{C}(3)$ for some $\hat{v}_0$ and this requires that $\diamond \in \hat{\rho}(y) \subseteq \hat{v}_0$ which contradicts $\hat{C}(3) = \emptyset$. For the slightly less precise analysis result where $\hat{\rho}$ and $\hat{C}$ are as above except that $\hat{\rho}(z) = \hat{C}(3) = \{\diamond\}$ we can indeed show that $(\hat{C},\hat{\rho}) \models_{\mathsf{cs}} (\lambda x.(x\,3)^1)(\lambda y.(\lambda z.y^3)^2) : \{\{\!|\lambda z.y^3|\!\}\}$. □

2.3 A Compositional Verbose Specification

To get an even more implementation oriented specification we shall extend our use of the cache to record the analysis estimates of *all* the subexpressions; we shall call the analysis *verbose* since the analysis domain captures all the analysis information of interest for the complete program. To formalise this we extend our notion of labelling to

$$e^{\ell} ::= c^{\ell} \mid x^{\ell} \mid (\lambda x_0.e_0^{\ell_0})^{\ell} \mid (e_1^{\ell_1}\, e_2^{\ell_2})^{\ell}$$

and as before the labels play no role in the semantics; they only serve as explicit reference points for the analysis and take a form resembling addresses in the syntax tree. The judgements now have the form

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^{\ell}$$

and the intension is that $\hat{C}(\ell)$ is the analysis estimate for e. The analysis is defined by the following clauses that are obtained by a simple rewriting of those of Subsection 2.2:

$$\begin{array}{rll}
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} c^{\ell} & \mathsf{iff} & \diamond \in \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} x^{\ell} & \mathsf{iff} & \hat{\rho}(x) \subseteq \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} (\lambda x_0.e_0^{\ell_0})^{\ell} & \mathsf{iff} & \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{C}(\ell) \ \wedge \\
 & & \hat{\rho}(x_0) \neq \emptyset \Rightarrow (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e_0^{\ell_0} \\
(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} (e_1^{\ell_1}\, e_2^{\ell_2})^{\ell} & \mathsf{iff} & (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e_1^{\ell_1} \ \wedge \ (\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e_2^{\ell_2} \ \wedge \\
 & & \forall \{\!|\lambda x_0.e_0^{\ell_0}|\!\} \in \hat{C}(\ell_1) : \hat{C}(\ell_2) \subseteq \hat{\rho}(x_0) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(\ell)
\end{array}$$

Note that in contrast to the previous two specifications all entities occurring on the right hand sides of the clauses also occur on the left hand side; hence there are no implicitly quantified variables.

Example 6. We now write the expression as $(\lambda x.(x^4\,3^5)^1)^6(\lambda y.(\lambda z.y^3)^2)^7)^8$ and an acceptable analysis estimate is

$$(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} (\lambda x.(x^4\,3^5)^1)^6(\lambda y.(\lambda z.y^3)^2)^7)^8$$

where $\hat{\rho}$ is as in Example 3 and:

$\hat{C} =$

1	2	3	4
$\{(\lambda z.y^3)\}$	$\{(\lambda z.y^3)\}$	$\emptyset$	$\{(\lambda y.(\lambda z.y^3)^2)\}$
5	6	7	8
$\{\diamond\}$	$\{(\lambda x.(x^4\ 3^5)^1)\}$	$\{(\lambda y.(\lambda z.y^3)^2)\}$	$\{(\lambda z.y^3)\}$

It is straightforward to verify that this is indeed an acceptable analysis estimate; in particular there is no need to make any guesses of intermediate judgements as all the required information is available in the cache. We omit the details. □

The specification is clearly *compositional* and we say that it is *verbose* since the cache $\hat{C}$ of a judgement $(\hat{C}, \hat{\rho}) \models_{\mathsf{cv}} e^\ell$ contains all the analysis information of interest for the program.

Relationship between the specifications. The verbose specification is a refinement of the specification of Subsection 2.2 in the sense that:

Lemma: If $(\hat{C}, \hat{\rho}) \models_{\mathsf{cv}} e^\ell$ then $(\hat{C}, \hat{\rho}) \models_{\mathsf{cs}} e : \hat{C}(\ell)$ (modulo the labelling of expressions).

Thus an implementation of the verbose specification will also provide solutions to the abstract (succinct as well as compositional) specification. The systematic transformation of a compositional succinct specification into a compositional verbose specification is studied in [21]. One may note that the converse of the lemma also holds due to the compositionality of the specifications.

2.4 An Abstract Verbose Specification

The verbose specification style is characterised by the explicit caching of analysis results for all subexpressions and we shall now combine it with the abstract specification style thereby generalising the previous specification to open systems. The new judgements have the form

$$(\hat{C}, \hat{\rho}) \models_{\mathsf{av}} e^\ell$$

where the labelling scheme is as in Subsection 2.3. The analysis is specified by the clauses:

$$\begin{aligned}
(\hat{C}, \hat{\rho}) \models_{\mathsf{av}} c^\ell \quad &\mathsf{iff} \quad \diamond \in \hat{C}(\ell) \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{av}} x^\ell \quad &\mathsf{iff} \quad \hat{\rho}(x) \subseteq \hat{C}(\ell) \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{av}} (\lambda x_0.e_0^{\ell_0})^\ell \quad &\mathsf{iff} \quad (\lambda x_0.e_0^{\ell_0}) \in \hat{C}(\ell) \\
(\hat{C}, \hat{\rho}) \models_{\mathsf{av}} (e_1^{\ell_1}\ e_2^{\ell_2})^\ell \quad &\mathsf{iff} \quad (\hat{C}, \hat{\rho}) \models_{\mathsf{av}} e_1^{\ell_1} \wedge (\hat{C}, \hat{\rho}) \models_{\mathsf{av}} e_2^{\ell_2} \wedge \\
&\qquad \forall (\lambda x_0.e_0^{\ell_0}) \in \hat{C}(\ell_1) : \hat{C}(\ell_2) \neq \emptyset \Rightarrow \\
&\qquad\quad [\hat{C}(\ell_2) \subseteq \hat{\rho}(x_0) \wedge (\hat{C}, \hat{\rho}) \models_{\mathsf{av}} e_0^{\ell_0} \wedge \hat{C}(\ell_0) \subseteq \hat{C}(\ell)]
\end{aligned}$$

Note again that λ-abstractions that are never applied need not have acceptable analysis estimates.

Example 7. For our example expression we now have

$$(\hat{C}, \hat{\rho}) \models_{\mathsf{av}} (\lambda x.(x^4\ 3^5)^1)^6 (\lambda y.(\lambda z.y^3)^2)^7)^8$$

where $\hat{C}$ and $\hat{\rho}$ are as in the previous example. It is straightforward to verify that this is indeed a valid judgement and as in Subsection 2.3 no guessing of intermediate judgements are needed as all the required information is available in the cache. We omit the details. □

Clearly the specification is *abstract* as the clause for application demands the analysis of all bodies of λ-abstractions that result from the operator part. It is also *verbose* since the abstract domains contain all the analysis information of interest for the program.

Relationship between the specifications. In analogy with the result of Subsection 2.2 (and those of [8,18]) we have:

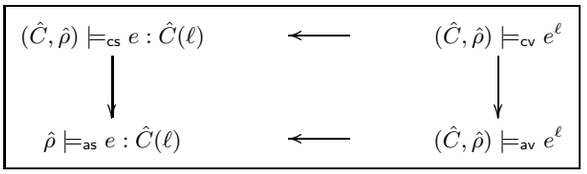
Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{cv}} e^{\ell}$ and $\hat{C}$ and $\hat{\rho}$ only mention λ-abstractions occurring in e then $(\hat{C},\hat{\rho}) \models_{\mathsf{av}} e^{\ell}$ (modulo the labelling of expressions).

We can also express a relationship between the two abstract specifications (analogously to the result stated in Subsection 2.3):

Lemma: If $(\hat{C},\hat{\rho}) \models_{\mathsf{av}} e^{\ell}$ then $\hat{\rho} \models_{\mathsf{as}} e : \hat{C}(\ell)$ (modulo the labelling of expressions).

The converse results do not hold in general due to the polyvariant nature of the abstract specifications.

The four results are summarised in the "flow logic square":



3 The Methodology of Flow Logic

A flow logic *specification* defines:

- the universe of discourse for the analysis estimates;
- the format of the judgements; and
- the defining clauses.

The universe of discourse usually is given by complete lattices and hence follows the approaches of data flow analysis, constraint-based analysis and abstract interpretation. The formulation of the judgements and their clauses focuses on *what* the analysis does and not *how* it does it. There are several benefits from this: (*i*) it is not necessary to think about the design and the implementation of the analysis at the same time; (*ii*) one can concentrate on specifying the analysis, i.e. on how to collect analysis information and link it together and (*iii*) the problem of trading efficiency for precision can be studied at the specification level and hence independently of implementation details.

Having specified a flow logic the next step is to show that the analysis enjoys a number of desirable properties:

- the judgements are well-defined;
- the judgements are semantically correct;
- the judgements have a Moore family (or model intersection) property; and
- the judgements have efficient implementations.

The Moore family property is important because it ensures not only that each program can be analysed but also that it has a best or most precise analysis result. We now survey the appropriate techniques and illustrate them on the example analyses of Section 2.

3.1 Well-Definedness of the Analysis

It is straightforward to see that the compositional specifications are well-defined; a simple induction on the syntax of the programs will do. However, it is not so obvious that the abstract specifications are well-defined; for the analysis specified in Subsection 2.1 it is the following clause that is problematic:

$$\begin{array}{ll}\hat{\rho} \models_{\mathsf{as}} e_1\, e_2 : \hat{v} & \mathsf{iff} \quad \hat{\rho} \models_{\mathsf{as}} e_1 : \hat{v}_1 \ \wedge\ \hat{\rho} \models_{\mathsf{as}} e_2 : \hat{v}_2 \ \wedge \\ & \qquad \forall \{\!|\lambda x_0.e_0|\!\} \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow [\hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge \hat{\rho} \models_{\mathsf{as}} e_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}]\end{array}$$

To ensure that an abstract specification is well-defined we shall turn the defining clauses into an appropriate functional over some complete lattice — in doing so we shall impose some demands on the form of the clauses such that monotonicity of the functional is enforced. The benefit is that then Tarski's fixed point theorem ensures that the functional has fixed points, in particular that it has a least as well as a greatest fixed point. Since we are giving meaning to a specification, the analysis is defined as the *greatest* fixed point (i.e. co-inductively) — intuitively, this means that only judgements that violates the conditions imposed by the specification are excluded. Actually, if we had insisted on defining the analysis as the least fixed point of the functional then there are important parts of the development that would fail; in particular, the Moore family property would not hold in general so there are programs that cannot be analysed [18].

The approach of defining the analysis coinductively works for abstract as well as compositional specifications; however, in the latter case it turns out that the functional has *exactly one* fixed point so the least and the greatest fixed points coincide and we may safely say that the analysis is defined *inductively* as mentioned above.

We shall illustrate the technique for the *abstract succinct specification* of Subsection 2.1. Write $(\mathbf{Q}, \sqsubseteq)$ for the complete lattice $\widehat{\mathbf{Env}} \times \mathbf{Exp} \times \widehat{\mathbf{Val}} \to \{\mathsf{tt}, \mathsf{ff}\}$ with the ordering

$$Q_1 \sqsubseteq Q_2 \quad \mathsf{iff} \quad \forall(\hat{\rho}, e, \hat{v}) : (Q_1(\hat{\rho}, e, \hat{v}) = \mathsf{tt}) \Rightarrow (Q_2(\hat{\rho}, e, \hat{v}) = \mathsf{tt})$$

expressing that Q_2 may give true more often than Q_1; it is easy to check that this indeed defines a complete lattice. The clauses for $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ define the following functional $\mathcal{Q}_{\mathsf{as}} : \mathbf{Q} \to \mathbf{Q}$ (i.e. $\mathcal{Q}_{\mathsf{as}} : \mathbf{Q} \to (\widehat{\mathbf{Env}} \times \mathbf{Exp} \times \widehat{\mathbf{Val}} \to \{\mathsf{tt}, \mathsf{ff}\})$):

$$
\begin{aligned}
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, c, \hat{v}) &= (\diamond \in \hat{v}) \\
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, x, \hat{v}) &= (\hat{\rho}(x) \subseteq \hat{v}) \\
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, \lambda x_0.e_0, \hat{v}) &= (\{\!|\lambda x_0.e_0|\!\} \in \hat{v}) \\
\mathcal{Q}_{\mathsf{as}}(Q)(\widehat{\rho}, e_1\, e_2, \hat{v}) &= \exists \hat{v}_1, \hat{v}_2 : Q(\widehat{\rho}, e_1, \hat{v}_1) \wedge Q(\widehat{\rho}, e_2, \hat{v}_2) \wedge \\
&\qquad \forall \{\!|\lambda x_0.e_0|\!\} \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow \\
&\qquad\qquad [\exists \hat{v}_0 : \hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge Q(\widehat{\rho}, e_0, \hat{v}_0) \wedge \hat{v}_0 \subseteq \hat{v}]
\end{aligned}
$$

It is straightforward to check that $\mathcal{Q}_{\mathsf{as}}$ is monotonic: when Q gives true more often then so does $\mathcal{Q}_{\mathsf{as}}(Q)$ — intuitively, the reason why $\mathcal{Q}_{\mathsf{as}}$ is monotonic is that all occurrences of Q on the right hand sides of the equations occur in positive positions: they are prefixed by an even (in particular none) number of negations. It then follows from Tarski's fixed point theorem that $\mathcal{Q}_{\mathsf{as}}$ has a complete lattice of fixed points in $\mathbf{Q}$ and we define $\models_{\mathsf{as}}$ to be the *greatest* fixed point.

3.2 Semantic Correctness

Flow logic is a *semantics based* approach to static analysis meaning that the information obtained from the analysis can be proved correct with respect to a semantics. The correctness can be established with respect to many different kinds of semantics (e.g. big-step or small-step operational semantics or denotational semantics); however, as pointed out in [15,16] the actual choice of semantics may significantly influence the style of the proof.

To establish semantic correctness we must define a *correctness relation* between the entities of the semantics and the analysis estimates. For our example both the semantics and the analysis operate with values and environments so we shall define two relations $\mathfrak{R}_{\mathsf{Val}} \subseteq \mathbf{Val} \times (\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}})$ and $\mathfrak{R}_{\mathsf{Env}} \subseteq \mathbf{Env} \times \widehat{\mathbf{Env}}$. The formal definitions amount to nothing but a formalisation of the intuition behind how the semantic entities are represented in the analysis:

$$
\begin{aligned}
c\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho}) \quad &\mathsf{iff} \quad \diamond \in \hat{v} \\
\langle \lambda x_0.e_0, \rho_0 \rangle\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho}) \quad &\mathsf{iff} \quad \{\!|\lambda x_0.e_0|\!\} \in \hat{v} \ \wedge\ \rho_0\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho} \\
\rho\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho} \quad &\mathsf{iff} \quad \forall x \in \mathsf{dom}(\rho) : \rho(x)\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{\rho}(x), \hat{\rho})
\end{aligned}
$$

It is immediate to show that the relations are well-defined by induction on the size of the semantic entities.

We shall now exploit that the abstract values and environments can be viewed as complete lattices in the following way: The abstract values $\widehat{\mathbf{Val}}$ constitutes a powerset and can naturally be equipped with the subset ordering; this ordering is then lifted in a pointwise manner to the abstract environments of $\widehat{\mathbf{Env}}$ and to the pairs of $\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}}$. Following the tradition of the denotational approach to abstract interpretation it is then desirable that the correctness relations satisfy admissibility conditions like

$$
\begin{aligned}
v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho}) \ \wedge\ (\hat{v}, \hat{\rho}) \sqsubseteq (\hat{v}', \hat{\rho}') \quad &\Rightarrow \quad v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}', \hat{\rho}') \\
(\forall (\hat{v}, \hat{\rho}) \in Y \subseteq (\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}}) : v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho})) \quad &\Rightarrow \quad v\ \mathfrak{R}_{\mathsf{Val}}\ (\bigsqcap Y)
\end{aligned}
$$

and similarly for $\mathfrak{R}_{\mathsf{Env}}$. The first condition expresses that a small abstract value is more precise than a larger one whereas the second condition expresses that

there always is a least abstract value for describing a value. In Subsection 3.3 we show that these properties can be lifted to the judgements.

Given a semantics that relates pairs of configurations, the correctness result is often expressed as a *subject reduction result*: if the first configuration is described by the analysis estimate then so is the second configuration. We shall first illustrate this for the abstract succinct analysis of Subsection 2.1 against the big-step operational semantics of Section 2 and subsequently we illustrate it against a small-step operational semantics presented below.

Big-step operational semantics. Recall that the transitions of the semantics have the form $\rho \vdash e \to v$ meaning that executing e in the environment ρ will give the value v. The correctness result for the analysis is expressed by:

Theorem: If $\rho \vdash e \to v$, $\rho\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho}$ and $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ then $v\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho})$.

So if $\hat{\rho}$ describes ρ then the result v of evaluating e in ρ is described by $\hat{v}$. In the special case where v is a closure $\langle \lambda x_0.e_0, \rho_0 \rangle$ this amounts to $(\!|\lambda x_0.e_0|\!) \in \hat{v}$ (and $\rho_0\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho}$) and in the case where v is a constant c it amounts to $\diamond \in \hat{v}$. The proof of the theorem is by induction in the inference $\rho \vdash e \to v$.

Small-step operational semantics. An environment-based small-step operational semantics for the λ-calculus uses values $v \in \mathbf{Val}$ together with intermediate expressions containing closures $\langle \lambda x_0.e_0, \rho_0 \rangle$ and special constructs of the form $\mathsf{bind}\,\rho_0\,\mathsf{in}\,e_0$; the role of the latter is to stack the environments arising in applications. The transitions are written as $\rho \Vdash e \to e'$ and are defined by:

$$\rho \Vdash x \to \rho(x) \qquad \frac{\rho \Vdash e_1 \to e_1'}{\rho \Vdash e_1\, e_2 \to e_1'\, e_2}$$

$$\rho \Vdash \lambda x_0.e_0 \to \langle \lambda x_0.e_0, \rho \rangle$$

$$\rho \Vdash (\langle \lambda x_0.e_0, \rho_0 \rangle)\, v \to \mathsf{bind}\,\rho_0[x_0 \mapsto v]\,\mathsf{in}\,e_0 \qquad \frac{\rho \Vdash e_2 \to e_2'}{\rho \Vdash e_1\, e_2 \to e_1\, e_2'}$$

$$\frac{\rho_0 \Vdash e \to e'}{\rho \Vdash \mathsf{bind}\,\rho_0\,\mathsf{in}\,e \to \mathsf{bind}\,\rho_0\,\mathsf{in}\,e'} \qquad \frac{\rho_0 \Vdash e \to v}{\rho \Vdash \mathsf{bind}\,\rho_0\,\mathsf{in}\,e \to v}$$

The correctness result will still be a subject reduction result and since some of the configurations may contain intermediate expressions we shall need to *extend the analysis* to handle these:

$$\hat{\rho} \models_{\mathsf{as}} \langle \lambda x_0.e_0, \rho_0 \rangle : \hat{v} \quad \mathsf{iff} \quad \langle \lambda x_0.e_0, \rho_0 \rangle\ \mathfrak{R}_{\mathsf{Val}}\ (\hat{v}, \hat{\rho})$$

$$\hat{\rho} \models_{\mathsf{as}} \mathsf{bind}\,\rho_0\,\mathsf{in}\,e_0 : \hat{v} \quad \mathsf{iff} \quad \hat{\rho} \models_{\mathsf{as}} e_0 : \hat{v}\ \wedge\ \rho_0\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho}$$

In both cases we use the correctness relations $\mathfrak{R}_{\mathsf{Val}}$ and $\mathfrak{R}_{\mathsf{Env}}$ to express the relationship between the entities of the semantics and the analysis. As before the analysis and the correctness relations are easily shown to be well-defined.

We can now formulate the semantic correctness of the analysis as

Theorem: If $\rho \Vdash e \to e'$, $\rho\ \mathfrak{R}_{\mathsf{Env}}\ \hat{\rho}$ and $\hat{\rho} \models_{\mathsf{as}} e : \hat{v}$ then $\hat{\rho} \models_{\mathsf{as}} e' : \hat{v}$.

expressing that the analysis estimate remains acceptable as the computation proceeds; the proof is by induction on the transitions of the semantics. This correctness result is slightly stronger than the previous one since it also applies to non-terminating computations.

Similar results can be obtained for the other three formulations of the analysis; naturally the correctness relations have to be extended to take the cache into account. We refer to [16] for a discussion of how the choice of semantics may influence the required proof techniques.

3.3 Moore Family Result

Having specified an analysis it is natural to ask whether every expression admits an acceptable analysis estimate and whether every expression has a best or most informative analysis estimate. To answer both of these questions in the affirmative it is sufficient to prove that the set of acceptable analysis estimates enjoys a Moore family (or model intersection) property: A *Moore family* is a subset $\widehat{V}$ of a complete lattice satisfying that whenever $Y \subseteq \widehat{V}$ then $\bigsqcap Y \in \widehat{V}$.

Let us once again consider the abstract succinct specification of Subsection 2.1. The complete lattice of interest is $\widehat{\mathbf{Val}} \times \widehat{\mathbf{Env}}$ equipped with the pointwise subset ordering and the Moore family result can be formulated as follows:

Theorem: The set $\{(\hat{\rho}, \hat{v}) \mid \hat{\rho} \models_{\mathsf{as}} e : \hat{v}\}$ is a Moore family for all e.

Taking $Y = \{(\hat{\rho}, \hat{v}) \mid \hat{\rho} \models_{\mathsf{as}} e : \hat{v}\}$ it follows that each expression has a least and acceptable analysis estimate.

Similar results can be formulated for the other styles of specification; the proof will be by co-induction or induction depending on whether the acceptability judgement is defined by co-induction or induction [18].

3.4 Efficient Implementation

The compositional verbose specification of Subsection 2.3 is a good starting point for actually obtaining an implementation. The idea will be to turn it into an algorithm for computing a set of conditional constraints that subsequently can be solved using standard constraint solvers as available in for example the BANE system [2].

For our example analysis we shall introduce two arrays C and R indexed by the labels and variables occurring in the expression $e_\star$ of interest and corresponding to the mappings $\hat{C}$ and $\hat{\rho}$ of the specification. The conditional constraints have the form *lst* => *lhs* $\subseteq$ *rhs* and express that if the conjunction of the conditions of the list *lst* holds then one set is included in another. The list may contain conditions of the form *set* $\neq \emptyset$ and *elm* $\in$ *set* with their obvious interpretation.

Initially, the algorithm $\mathcal{C}_{\mathsf{cv}}$ is applied to $e_\star$ and an empty list ϵ of conditions and it will construct a set of conditional constraints. The general algorithm is as follows:

$$
\begin{aligned}
\mathcal{C}_{\mathsf{cv}}[\![c^\ell]\!]lst &= \{\ lst \texttt{ => } \{\diamond\} \subseteq \mathsf{C}[\ell]\ \} \\
\mathcal{C}_{\mathsf{cv}}[\![x^\ell]\!]lst &= \{\ lst \texttt{ => } \mathsf{R}[x] \subseteq \mathsf{C}[\ell]\ \} \\
\mathcal{C}_{\mathsf{cv}}[\![(\lambda x_0.e_0^{\ell_0})^\ell]\!]lst &= \{\ lst \texttt{ => } \{\lambda x_0.e_0^{\ell_0}\} \subseteq \mathsf{C}[\ell]\ \} \ \cup\ \mathcal{C}_{\mathsf{cv}}[\![e_0^{\ell_0}]\!](lst\,\hat{}\,(\mathsf{R}[x_0] \neq \emptyset)) \\
\mathcal{C}_{\mathsf{cv}}[\![(e_1^{\ell_1}\ e_2^{\ell_2})^\ell]\!]lst &= \mathcal{C}_{\mathsf{cv}}[\![e_1^{\ell_1}]\!]lst \ \cup\ \mathcal{C}_{\mathsf{cv}}[\![e_2^{\ell_2}]\!]lst \\
&\quad \cup \{\ lst\,\hat{}\,(\{\lambda x_0.e_0^{\ell_0}\} \in \mathsf{C}[\ell_1]) \texttt{ => } \mathsf{C}[\ell_2] \subseteq \mathsf{R}[x_0] \ \mid \lambda x_0.e_0^{\ell_0} \text{ is in } e_\star\} \\
&\quad \cup \{\ lst\,\hat{}\,(\{\lambda x_0.e_0^{\ell_0}\} \in \mathsf{C}[\ell_1]) \texttt{ => } \mathsf{C}[\ell_0] \subseteq \mathsf{C}[\ell] \ \mid \lambda x_0.e_0^{\ell_0} \text{ is in } e_\star\}
\end{aligned}
$$

For each λ-abstraction a new condition is appended to the list; the current list is then used as the condition whenever a constraint is generated. For each application we construct a *set* of conditional constraints, one for each of the

λ-abstractions of $e_\star$. In this simple case, the set of generated constraints can be solved by a simple worklist algorithm.

One can formally prove a *syntactic soundness and completeness result* expressing that any solution to the above constraint system is also an acceptable analysis estimate according to the specification of Subsection 2.3 and vice versa:

Lemma: $(\hat{C}, \hat{\rho}) \models_{\mathsf{cv}} e_\star$ if and only if $(\hat{\rho}, \hat{C})$ satisfies the constraints of $\mathcal{C}_{\mathsf{cv}}[\![e_\star]\!]\epsilon$.

Hence it follows that a solution $(\hat{\rho}, \hat{C})$ to the constraints also will be an acceptable analysis result for the other three specifications in Section 2. We refer to [18] for further details.

4 Pragmatics

The aim of this section is to illustrate that the flow logic approach scales up to other programming paradigms. Space does not allow us to give full specifications so we shall only present fragments of specifications and refer to other papers for more details. We first consider examples from the imperative, the object-oriented and the concurrent paradigm. Then we show how the ideas can be combined for multi-paradigmatic languages. We shall mainly consider abstract succinct specifications; similar developments can be performed for the other styles of specifications.

4.1 Imperative Constructs

Imperative languages are characterised by the presence of a *state* that is updated as the execution proceeds; typical constructs are assignments and sequencing of statements:

$$S ::= x := e \mid S_1; S_2 \mid \cdots$$

Here $x \in \mathbf{Var}$ and $e \in \mathbf{Exp}$ are unspecified sets of variables and expressions. We assume a standard semantics operating over states $\sigma \in \mathbf{St}$.

The analysis. We shall consider a forward analysis using a complete lattice $(\widehat{\mathbf{St}}, \sqsubseteq)$ of abstract states and with transfer functions $\phi_{x:=e} : \widehat{\mathbf{St}} \to \widehat{\mathbf{St}}$ specifying how the assignments $x := e$ modify the abstract states; as an example we may have $\widehat{\mathbf{St}} = \mathbf{Var} \to \widehat{\mathbf{Val}}$ and $\phi_{x:=e}(\widehat{\sigma}) = \widehat{\sigma}[x \mapsto \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma}]$ where $\widehat{\mathcal{E}}$ defines the analysis of expressions. The judgements of the analysis have the form

$$\models S : \widehat{\sigma} \rhd \widehat{\sigma}'$$

and expresses that $\widehat{\sigma} \rhd \widehat{\sigma}'$ is an acceptable analysis estimate for S meaning that if $\widehat{\sigma}$ describes the initial state then $\widehat{\sigma}'$ will describe the possible final states. For the two constructs above we have the clauses:

$$\models x := e : \widehat{\sigma} \rhd \widehat{\sigma}' \quad \text{iff} \quad \phi_{x:=e}(\widehat{\sigma}) \sqsubseteq \widehat{\sigma}'$$

$$\models S_1; S_2 : \widehat{\sigma} \rhd \widehat{\sigma}'' \quad \text{iff} \quad \models S_1 : \widehat{\sigma} \rhd \widehat{\sigma}' \wedge \models S_2 : \widehat{\sigma}' \rhd \widehat{\sigma}''$$

The first clause expresses that the result $\phi_{x:=e}(\widehat{\sigma})$ of analysing the assignment has been captured by $\widehat{\sigma}'$. The second clause expresses that in order for $\widehat{\sigma} \rhd \widehat{\sigma}''$ to be an acceptable analysis estimate for $S_1; S_2$ there must exist an abstract state $\widehat{\sigma}'$ such that $\widehat{\sigma} \rhd \widehat{\sigma}'$ is an acceptable analysis estimate for S_1 and $\widehat{\sigma}' \rhd \widehat{\sigma}''$ is an acceptable analysis estimate for S_2.

A coarse analysis. For later reference we shall present a coarser analysis that does not distinguish between the program points and hence uses a single abstract state $\widehat{\sigma} \in \widehat{\mathbf{St}} = \mathbf{Var} \to \widehat{\mathbf{Val}}$ to capture the information about *all* the states that may occur during execution. The judgements of this (verbose) analysis have the form

$$\widehat{\sigma} \models' S$$

and for our two constructs it is defined by the fairly straightforward clauses:

$$\begin{array}{lll} \widehat{\sigma} \models' x := e & \mathsf{iff} & \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma} \sqsubseteq \widehat{\sigma}(x) \\ \widehat{\sigma} \models' S_1; S_2 & \mathsf{iff} & \widehat{\sigma} \models' S_1 \ \wedge \ \widehat{\sigma} \models' S_2 \end{array}$$

4.2 Object-Oriented Constructs

To illustrate the ideas on an imperative object-oriented language we use a fragment of Abadi and Cardelli's imperative object calculus [1]:

$$O ::= [m_i = \varsigma(x_i).O_i]_{i=1}^n \mid O.m \mid O.m := \varsigma(x_0).O_0 \mid \cdots$$

Here $[m_i = \varsigma(x_i).O_i]_{i=1}^n$ introduces an object with ordered components defining methods with (distinct) names $m_1, \cdots, m_n \in \mathbf{Nam}$, formal parameters $x_1, \cdots, x_n \in \mathbf{Var}$ and bodies $O_1, \cdots, O_n$. Method invocation $O.m$ amounts to first evaluating O to determine an object o of the form $[m_i = \varsigma(x_i).O_i]_{i=1}^n$ and then evaluating the body O_j of the method m_j associated with m (i.e. $m = m_j$) while binding the formal parameter x_j to o so as to allow self-reference; the result of this will then be returned as the overall value. The construct $O.m := \varsigma(x_0).O_0$ will update the method m of the object o that O evaluates to and hence it affects the global state; it is required that o already has a method named m. We omit the formal semantics.

The analysis. The aim is to determine the set of objects that can reach various points in the program. An object $[m_i = \varsigma(x_i).O_i]_{i=1}^n$ will be represented by a tuple $\overrightarrow{m} = (m_1, \cdots, m_n) \in \mathbf{Nam}^\star$ listing the names of its methods and a method will be represented by an abstract closure $\{\!|\varsigma(x_0).O_0|\!\} \in \mathbf{Mt}$. The judgements of the analysis have the form

$$(\hat{\rho}, \widehat{\sigma}) \models O : \hat{v}$$

and expresses the acceptability of the analysis estimate $\hat{v}$ for O in the context described by $\hat{\rho}$ and $\widehat{\sigma}$. As in Section 2 the idea is that $\hat{v} \in \widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Nam}^\star)$ describes the set of abstract objects that O may evaluate to and $\hat{\rho} : \mathbf{Var} \to \widehat{\mathbf{Val}}$ describes the abstract values associated with the variables. For simplicity we

follow the coarser approach of Subsection 4.1 and represent *all* the states that may arise during the computation by a single abstract state $\widehat{\sigma} \in \widehat{\mathbf{St}} = (\mathbf{Nam}^\star \times \mathbf{Nam}) \to \mathcal{P}(\mathbf{Mt})$ that for each abstract object and method name determines a set of abstract closures.

The clauses for object definition and method call are very similar to the clauses for λ-abstraction and function application in Subsection 2.1:

$$\begin{array}{rcll}(\hat{\rho},\widehat{\sigma}) \models [m_i = \varsigma(x_i).O_i]_{i=1}^n : \hat{v} & \mathsf{iff} & (m_1,\cdots,m_n) \in \hat{v} \wedge \\ & & \forall i \in \{1,\cdots,n\} : \{\!|\varsigma(x_i).O_i|\!\} \in \widehat{\sigma}((m_1,\cdots,m_n),m_i) \\ (\hat{\rho},\widehat{\sigma}) \models O.m : \hat{v} & \mathsf{iff} & (\hat{\rho},\widehat{\sigma}) \models O : \hat{v}' \wedge \\ & & \forall \overrightarrow{m} \in \hat{v}', \forall \{\!|\varsigma(x_0).O_0|\!\} \in \widehat{\sigma}(\overrightarrow{m},m) : \\ & & \quad \overrightarrow{m} \in \hat{\rho}(x_0) \wedge (\hat{\rho},\widehat{\sigma}) \models O_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}\end{array}$$

The first clause expresses that $(m_1,\cdots,m_n)$ represents a possible value of the construct and ensures that the abstract state $\widehat{\sigma}$ contains the relevant abstract closures. The second clause first requires that the object O has an acceptable analysis estimate $\hat{v}'$. Then it inspects each of the abstract closures $\{\!|\varsigma(x_0).O_0|\!\}$ that may be associated with m for some abstract object $\overrightarrow{m}$ of $\hat{v}'$: it is required that $\hat{\rho}$ records that the formal parameter x_0 may be bound to the actual object $\overrightarrow{m}$ and that the body O_0 of the method has an acceptable analysis estimate that is included in the overall analysis estimate.

The clause for method update is inspired by the clause for assignment in the coarser analysis of Subsection 4.1:

$$\begin{array}{rcll}(\hat{\rho},\widehat{\sigma}) \models O.m := \varsigma(x_0).O_0 : \hat{v} & \mathsf{iff} & (\hat{\rho},\widehat{\sigma}) \models O : \hat{v}' \wedge \\ & & \forall \overrightarrow{m} \in \hat{v}' : \widehat{\sigma}(\overrightarrow{m},m) \neq \emptyset \Rightarrow \\ & & \qquad \overrightarrow{m} \in \hat{v} \ \wedge \ \{\!|\varsigma(x_0).O_0|\!\} \in \widehat{\sigma}(\overrightarrow{m},m)\end{array}$$

It expresses that in order for $\hat{v}$ to be an acceptable analysis estimate for the assignment then it must be the case that O has an acceptable analysis estimate $\hat{v}'$. Furthermore, for each abstract object of $\hat{v}'$ that has a method named m it is required that the abstract state $\widehat{\sigma}$ records the abstract closure $\{\!|\varsigma(x_0).O_0|\!\}$.

4.3 Concurrency

To illustrate how concurrency and communication can be handled we shall study a fragment of the π-calculus [12]:

$$P ::= \overline{u}\, t.P \mid u(x).P \mid (\nu\, n)P \mid P_1 \shuffle P_2 \mid \cdots$$

Here t, u are terms that can be either variables $x \in \mathbf{Var}$ or channel names $n \in \mathbf{Ch}$. The process $\overline{u}\, t.P$ will output the channel that t evaluates to over the channel that u evaluates to and then it will continue as P. The process $u(x).P$ will input a channel over the channel that u evaluates to, bind it to the variable x and then it will continue as the process P. The construct $(\nu\, n)P$ creates a new channel n with scope P. The two processes P_1 and P_2 of the parallel composition $P_1 \shuffle P_2$ act independently of one another but can also communicate when one of them performs an input and the other an output action over the same channel. We omit the detailed semantics.

The analysis. The aim will be to determine which channels may be communicated over which other channels; in [3] we show how this information can be used to validate certain security properties. The judgements of the analysis have the form

$$(\hat{\rho}, \widehat{\kappa}) \models P$$

and expresses that P has an acceptable analysis estimate in the context described by $\hat{\rho}$ and $\widehat{\kappa}$. Here $\hat{\rho} : \mathbf{Var} \to \mathcal{P}(\mathbf{Ch})$ maps the variables to the sets of channels they may be bound to and $\widehat{\kappa} : \mathbf{Ch} \to \mathcal{P}(\mathbf{Ch})$ maps the channels to the sets of channels that may be communicated over them. When formulating the analysis below we shall extend $\hat{\rho}$ to operate on terms and define $\hat{\rho}(n) = \{n\}$ for all channel names n. For the above constructs we define the analysis by the following clauses:

$$\begin{aligned}
(\hat{\rho}, \widehat{\kappa}) \models \overline{u}\, t.P \quad &\text{iff} \quad (\hat{\rho}, \widehat{\kappa}) \models P \ \wedge\ \forall n \in \hat{\rho}(u) : \hat{\rho}(t) \subseteq \widehat{\kappa}(n) \\
(\hat{\rho}, \widehat{\kappa}) \models u(x).P \quad &\text{iff} \quad (\hat{\rho}, \widehat{\kappa}) \models P \ \wedge\ \forall n \in \hat{\rho}(u) : \widehat{\kappa}(n) \subseteq \hat{\rho}(x) \\
(\hat{\rho}, \widehat{\kappa}) \models (\nu\, n)P \quad &\text{iff} \quad (\hat{\rho}, \widehat{\kappa}) \models P \\
(\hat{\rho}, \widehat{\kappa}) \models P_1 \parallel P_2 \quad &\text{iff} \quad (\hat{\rho}, \widehat{\kappa}) \models P_1 \ \wedge\ (\hat{\rho}, \widehat{\kappa}) \models P_2
\end{aligned}$$

The clause for output first ensures that the continuation P has an acceptable analysis estimate and it then requires that all the channels that y may be bound to also are recorded in the communication cache for all the channels that x may evaluate to. Turning to input it first ensures that the continuation has an acceptable analysis estimate and then it requires that the set of channels possibly bound to y includes those that may be communicated over all the channels that x may evaluate to. The clause for channel creation just requires that the subprocess has an acceptable analysis estimate. The final clause expresses that in order to have an acceptable analysis estimate for parallel composition we must have acceptable analysis estimates for the two subprocesses.

The above specification does not take into account that the semantics of the π-calculus allows α-renaming of names and variables; we refer to [3] for how to deal with this.

4.4 Combining Paradigms

We shall conclude by illustrating how the above techniques can be combined for multi-paradigmatic languages. Our example language will be Concurrent ML [24] that is an extension of Standard ML [13] and that combines the functional, imperative and concurrent paradigms. We shall consider the following fragment:

$$\begin{aligned}
e ::= {} & c \mid x \mid \texttt{fn}\ x_0\ \texttt{=>}\ e_0 \mid \texttt{fun}\ f\ x_0\ \texttt{=>}\ e_0 \mid e_1\, e_2 \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \cdots \\
& \mid e_1\ \texttt{:=}\ e_2 \mid e_1; e_2 \mid \texttt{let}\ x\ \texttt{=ref}_r\ \texttt{in}\ e \mid \texttt{deref}\ e \mid \cdots \\
& \mid \texttt{send}\ e_1\ \texttt{on}\ e_2 \mid \texttt{receive}\ e \mid \texttt{let}\ x\ \texttt{=chan}_n\ \texttt{in}\ e \mid \texttt{spawn}\ e \mid \cdots
\end{aligned}$$

The functional paradigm is represented by the first line where we have an extension of the λ-calculus with constructs for defining recursive functions and conditional expressions. The imperative constructs extend those of Subsection 4.1 by adding constructs for the dynamic creation of reference cells and for dereferencing a reference cell. The third line adds concurrency constructs similar to those of Subsection 4.3 except that parallelism is introduced by an explicit

`spawn`-construct. The labels $r \in \mathbf{Ref}$ and $n \in \mathbf{Ch}$ denote program points and are merely introduced to aid the analysis; they will be used to refer to the reference cells and channels created at the particular program points.

The analysis. As in the previous examples the aim will be to estimate which values an expression may evaluate to. The judgements have the form

$$(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e : \hat{v}$$

and expresses the acceptability of the analysis result $\hat{v}$ in the context of $\hat{\rho}$, $\widehat{\sigma}$ and $\widehat{\kappa}$. The abstract environment $\hat{\rho} : \mathbf{Var} \to \widehat{\mathbf{Val}}$ will record the abstract values that may be bound to the variables, the abstract state $\widehat{\sigma} : \mathbf{Ref} \to \widehat{\mathbf{Val}}$ will record the abstract values that may be bound to the reference cells and the mapping $\widehat{\kappa} : \mathbf{Ch} \to \widehat{\mathbf{Val}}$ will record the abstract values that may be communicated over the channels. In addition to the constant $\diamond$ and the abstract closures $\langle\!|\texttt{fn } x_0 \texttt{ => } e_0|\!\rangle$, the abstract values can now also be recursive abstract closures $\langle\!|\texttt{fun } f\ x_0 \texttt{ => } e_0|\!\rangle$, reference cells r and channels n.

For the functional constructs we adapt the clauses of Subsection 2.1:

$$\begin{array}{rll}
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models c : \hat{v} & \text{iff} & \diamond \in \hat{v} \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models x : \hat{v} & \text{iff} & \hat{\rho}(x) \subseteq \hat{v} \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models \texttt{fn } x_0 \texttt{ => } e_0 : \hat{v} & \text{iff} & \langle\!|\texttt{fn } x_0 \texttt{ => } e_0|\!\rangle \in \hat{v} \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models \texttt{fun } f\ x_0 \texttt{ => } e_0 : \hat{v} & \text{iff} & \langle\!|\texttt{fun } f\ x_0 \texttt{ => } e_0|\!\rangle \in \hat{v} \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1\ e_2 : \hat{v} & \text{iff} & (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1 : \hat{v}_1 \ \wedge\ (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_2 : \hat{v}_2 \ \wedge \\
 & & \forall \langle\!|\texttt{fn } x_0 \texttt{ => } e_0|\!\rangle \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow \\
 & & \quad [\hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}] \wedge \\
 & & \forall \langle\!|\texttt{fun } f\ x_0 \texttt{ => } e_0|\!\rangle \in \hat{v}_1 : \hat{v}_2 \neq \emptyset \Rightarrow \\
 & & \quad [\langle\!|\texttt{fun } f\ x_0 \texttt{ => } e_0|\!\rangle \in \hat{\rho}(f) \wedge \hat{v}_2 \subseteq \hat{\rho}(x_0) \wedge \\
 & & \qquad (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_0 : \hat{v}_0 \wedge \hat{v}_0 \subseteq \hat{v}] \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 : \hat{v} & \text{iff} & (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_0 : \hat{v}_0 \wedge \\
 & & \diamond \in \hat{v}_0 \Rightarrow [(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1 : \hat{v}_1 \wedge \hat{v}_1 \sqsubseteq \hat{v} \wedge \\
 & & \quad (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_2 : \hat{v}_2 \wedge \hat{v}_2 \sqsubseteq \hat{v}]
\end{array}$$

Recursion is handled much as ordinary function abstraction and the clause for application is extended to take care of recursion as well. The construct for conditional only requires that the branches have acceptable analysis estimates if the test could evaluate to a constant. Note that $\widehat{\sigma}$ and $\widehat{\kappa}$ play no role in this part of the specification.

For the imperative constructs we follow the approach of the coarser analysis of Subsection 4.1 and take:

$$\begin{array}{rll}
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1 \texttt{ := } e_2 : \hat{v} & \text{iff} & (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1 : \hat{v}_1 \ \wedge\ (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_2 : \hat{v}_2 \ \wedge \\
 & & \diamond \in \hat{v} \ \wedge\ \forall r \in \hat{v}_1 : \hat{v}_2 \subseteq \widehat{\sigma}(r) \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models \texttt{deref } e : \hat{v} & \text{iff} & (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e : \hat{v}' \ \wedge\ \forall r \in \hat{v}' : \widehat{\sigma}(r) \subseteq \hat{v} \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models \texttt{let } x \texttt{ = ref}_r \texttt{ in } e : \hat{v} & \text{iff} & r \in \hat{\rho}(x) \ \wedge\ (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e : \hat{v} \\
(\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1; e_2 : \hat{v} & \text{iff} & (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_1 : \hat{v}' \ \wedge\ (\hat{\rho}, \widehat{\sigma}, \widehat{\kappa}) \models e_2 : \hat{v}
\end{array}$$

The first clause reflects that an assignment in Standard ML evaluates to a dummy constant. In the second clause we consult the abstract state to determine the abstract values that the reference cell may contain. The `let`-construct evaluates to its body so we only ensure that the new reference cell (called r) is a potential value for the variable x. The final clause records that $e_1; e_2$ evaluates to the value of e_2.

Finally, for the concurrency constructs we take:

$$(\hat{\rho},\hat{\sigma},\hat{\kappa}) \models \texttt{send}\ e_1\ \texttt{on}\ e_2 : \hat{v} \quad \text{iff} \quad (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e_1 : \hat{v}_1\ \wedge\ (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e_2 : \hat{v}_2\ \wedge\ \diamond \in \hat{v}\ \wedge\ \forall n \in \hat{v}_2 : \hat{v}_1 \subseteq \hat{\kappa}(n)$$

$$(\hat{\rho},\hat{\sigma},\hat{\kappa}) \models \texttt{receive}\ e : \hat{v} \quad \text{iff} \quad (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e : \hat{v}'\ \wedge\ \forall n \in \hat{v}' : \hat{\kappa}(n) \subseteq \hat{v}$$

$$(\hat{\rho},\hat{\sigma},\hat{\kappa}) \models \texttt{let}\ x = \texttt{chan}_n\ \texttt{in}\ e : \hat{v} \quad \text{iff} \quad n \in \hat{\rho}(x)\ \wedge\ (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e : \hat{v}$$

$$(\hat{\rho},\hat{\sigma},\hat{\kappa}) \models \texttt{spawn}\ e : \hat{v} \quad \text{iff} \quad \diamond \in \hat{v}\ \wedge\ (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e : \hat{v}'\ \wedge\ \forall \langle\!|\texttt{fn}\ x_0 \texttt{=>} e_0|\!\rangle \in \hat{v}' : \diamond \subseteq \hat{\rho}(x_0) \wedge (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e_0 : \hat{v}_0\ \wedge\ \forall \langle\!|\texttt{fun}\ f\ x_0 \texttt{=>} e_0|\!\rangle \in \hat{v}' : \langle\!|\texttt{fun}\ f\ x_0 \texttt{=>} e_0|\!\rangle \in \hat{\rho}(f) \wedge \diamond \subseteq \hat{\rho}(x_0) \wedge (\hat{\rho},\hat{\sigma},\hat{\kappa}) \models e_0 : \hat{v}_0$$

The first clause reflects that the abstract values of e_1 represent potential values to be communicated over the channels that e_2 may evaluate to; the `send`-construct itself evaluates to a dummy constant. The second clause then records that the `receive`-construct may evaluate to the abstract values that may be communicated over the channels that e evaluates to. The `let`-construct is analogous to the `let`-construct for reference cells. The `spawn`-construct expects a function as parameter and, when executed, it will bind a dummy constant to the formal parameter and spawn a process for executing the body. In the analysis this is recorded using the same technique as we introduced for function application.

In [14,21,17] we present more precise analyses of the functional and imperative subset of the language and in [8] we give a more precise treatment of the functional and concurrent subset of the language.

5 Conclusion

We have shown that the flow logic approach applies to a variety of programming paradigms. The ease with which this can be done is mainly due to the *high abstraction level* supported by the approach: there is a clear separation between specifying an analysis and implementing it. Another important aspect is that even at the specification level one can combine different styles of specification for different aspects of the language and different aspects of the analysis.

The flow logic approach is also very flexible when it comes to integrating state-of-the-art techniques developed from data flow analysis, constraint-based analysis and abstract interpretation. In [14] we show how the approach of abstract interpretation can be combined with a constraint-based formulation of a control flow analysis for a functional language and the relationship to a number

of standard approaches like the k-CFA approach is clarified. This work is carried one step further in [17,21] where we extend the language to have functional as well as imperative constructs and show how to integrate classical techniques for interprocedural data flow analysis (call strings and assumption sets) into the specification; we also show how the standard technique of reference counts can be incorporated.

In the context of the ambient calculus [5] we have shown how powerful abstraction techniques based on tree grammars can be used in conjunction with flow logic [22] and in [20] we show how the three valued logic approach originally developed for analysing pointers in an imperative language [25] can be integrated with the flow logic approach.

Acknowledgement. The paper was written while the authors were visiting Universität des Saarlandes and Max-Planck Institute für Informatik, Saarbrücken, Germany.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proc. TIC'98*, 1998.
3. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the π-calculus with applications to security. *Information and Computation*, to appear, 2000.
4. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proc. PaCT'01*, number 2127 in Lecture Notes in Computer Science, pages 27–41. Springer-Verlag, 2001.
5. L. Cardelli and A. Gordon. Mobile ambients. In *Proc. FoSSaCS'98*, 1998.
6. D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. In *Proc. L & FP*, 1986.
7. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
8. K.L.S. Gasser, F. Nielson, and H. Riis Nielson. Systematic realisation of control flow analyses for CML. In *Proc. ICFP'97*, pages 38–51. ACM Press, 1997.
9. M. S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.
10. N. Heintze. Set-based analysis of ML programs. In *Proc. LFP '94*, pages 306–317, 1994.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
13. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
14. F. Nielson and H. Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.
15. F. Nielson and H. Riis Nielson. The flow logic of imperative objects. In *Proc. MFCS'98*, number 1450 in Lecture Notes in Computer Science, pages 220–228. Springer-Verlag, 1998.

16. F. Nielson and H. Riis Nielson. Flow logics and operational semantics. *Electronic Notes of Theoretical Computer Science*, 10, 1998.
17. F. Nielson and H. Riis Nielson. Interprocedural control flow analysis. In *Proc. ESOP'99*, number 1576 in Lecture Notes in Computer Science, pages 20–39. Springer-Verlag, 1999.
18. F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
19. F. Nielson, H. Riis Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In *Proc. CONCUR'99*, number 1664 in Lecture Notes in Computer Science, pages 463–477. Springer-Verlag, 1999.
20. F. Nielson, H. Riis Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In *Proc. ESOP'00*, number 1782 in Lecture Notes in Computer Science, pages 304–319. Springer-Verlag, 2000.
21. H. Riis Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. CC'98*, number 1383 in Lecture Notes in Computer Science, pages 109–127. Springer-Verlag, 1998.
22. H. Riis Nielson and F. Nielson. Shape analysis for mobile ambients. In *Proc. POPL'00*, pages 142–154. ACM Press, 2000.
23. G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
24. J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
25. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. POPL'99*, 1999.