Flow Logic and Operational Semantics

Flemming Nielson

Computer Science Department
University of Aarhus
DK-8000 Aarhus C, Denmark

Hanne Riis Nielson

Computer Science Department
University of Aarhus
DK-8000 Aarhus C, Denmark

Abstract
Flow logic is a “fast prototyping” approach to program analysis that shows great promise of being able to deal with a wide variety of languages and calculi for computation. However, seemingly innocent choices in the flow logic as well as in the operational semantics may inhibit proving the analysis correct. Our main conclusion is that environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences (like alpha-renaming).

1 Introduction
Flow logic facilitates the specification of program analyses [10] that automatically predict properties of programs holding in all executions. It allows to deal with a wide variety of languages; examples include the lambda-calculus with side-effects (a fragment of Standard ML) or communications (a fragment of Concurrent ML), several object based calculi, and a process algebra (the \(\pi\)-calculus). Analyses may be described in a succinct form (akin to program logic) or in a more verbose form (taking the form of constraint satisfaction); also they may be described at an abstract level of reasoning (using coinductive techniques) or in a more compositional manner (using inductive techniques). This allows to use the approach to first sketch the analysis, next refine it and prove it correct, and finally obtain an efficient implementation; furthermore, the development may be firmly rooted upon existing program analysis technology and insights, rather than having to start from scratch.

Structural operational semantics similarly allows to deal with a wide variety of languages. There are many choices that needs to be made concerning...
how to define the semantics: e.g. having small-step or big-step transitions, using environments or performing direct substitution, making use of evaluation contexts or having explicit rules for reduction in context. Many of these choices are seemingly innocent in the sense that they do not affect the “meaning” of the language being defined; and indeed different formulations of the semantics can often be proved equivalent (although the proofs are sometimes quite laborious).

This might suggest that one could deal with a new language or calculus in the following way: first the syntax and informal meaning is defined, then the program analysis is developed simultaneously with the operational semantics, and finally they are consolidated with respect to one another (and in particular the analysis is proved correct). One advantage of this approach is that the fine details of the language definition are consolidated not only by semantic considerations but also by more pragmatic considerations concerning the ease with which programs can be validated not to have anomalous behaviour; we believe that this is a key issue in designing languages that are both theoretically well-behaved and pragmatically useful. Another advantage is that the methods would then be more likely to scale up to “real life” languages because different teams of researchers could be responsible for different aspects of the development; this is a major parameter for the success of formal methods in software engineering and is often neglected in purely theoretical studies.

In our experience the above approach is fraught with problems. One reason is that the structure, size and complexity of the correctness proofs depend on characteristica of the flow logic (e.g. whether it is abstract or compositional) as well as on characteristica of the operational semantics (e.g. whether it uses environments or direct substitution). In some cases the choices may “contradict” one another so that no proof of correctness is possible and the analysis or semantics has to be changed. It is therefore important to identify general guidelines concerning what complications are likely to arise for what combinations – in order that the use of formal methods in this area may become a craft rather than a (black) art.

Our main conclusion is that environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences (like alpha-renaming) in terms of being able to accomodate a variety of specification styles for program analysis.

2 Setting the Scene

For simplicity this paper concentrates on an untyped lambda-calculus although analogous considerations apply to the more advanced object based and concurrent calculi mentioned above. The pure untyped lambda-calculus has the following syntax:

\[ e \in \text{Exp} \]
\[ e ::= x \mid \text{fn } x \Rightarrow e \mid e \; e \]
\[ x \in \text{Var} \]

where \( \text{Var} \) is an unspecified countably infinite set.

The succinct formulations of flow logic considered here do not require that all program points are made explicit; so there is no need to place explicit labels on all subexpressions (as in [9]) or to convert programs into “A-normal-form” (as in [4]). Instead we shall assume that all function abstractions have initially been alpha-renamed so as to have distinct formal parameters that are also disjoint from the set of global variables, \( \text{FV}(e_i) \), of the program of interest, \( e_i \).

Often program analysis is formulated as the demand to compute “the best” (or least) analysis information, \( \hat{\rho} \), that pertains to a program, \( e_i \): \( \hat{\rho} = A(e_i) \). Here we shall take the more flexible approach that a piece of analysis information, \( \hat{\rho} \), needs to be validated with respect to the program, \( e_i \): \( \hat{\rho} = e_i \) (yielding \( \text{tt} \) or \( \text{ff} \)). On the one hand this allows to develop algorithms for computing “the best” analysis information \[11,5\] and on the other hand it offers promise of analysing not only closed systems: whenever new expressions emerge from the environment it can be checked whether or not the current analysis has duly recorded all the possible effects of these expressions. The ability to analyse open systems is particularly important for calculi and languages dealing with distribution and mobility of software.

**Example 2.1** To give an example of the analysis consider the following simple program, \( e_i \), (in a slight extension of the syntax):

\[
\text{letrec } g = (\fn x => (g g))
\in g (\fn y => y)
\]

Here a function \( g \) is defined that ignores its parameter and calls itself recursively upon itself; the function is then called with the identity function as parameter.

We shall next consider an analysis that is called a control flow analysis \[15,16\], a closure analysis \[12,14\] or a set based analysis \[6\]. To do so we first define the abstract environment \( \hat{\rho} \) by:

\[
\hat{\rho}(g) = \{ \fn x => (g g) \}
\hat{\rho}(x) = \{ \fn x => (g g), \fn y => y \}
\hat{\rho}(y) = \emptyset
\]

The analysis of the program, \( e_i \), then amounts to a judgement of the form

\[
\hat{\rho} \vdash e_i : \emptyset
\]

saying that it will be correct to stipulate that \( g \) is bound to the recursive function itself, that the formal parameter \( x \) is bound to the recursive function or the identity function, that the formal parameter \( y \) is bound to nothing (corresponding to the identity function never being called), and that the program never returns any function (corresponding to the fact that it loops forever). To validate the analysis we will encounter other judgements like

\[
\hat{\rho} \vdash \fn x => (g g) : \{ \fn x => (g g) \}
\]
and to make this precise we need to clarify how the judgements, \( \hat{\rho} \models e : W \), are defined; this will be the subject of Section 4 after having defined the operational semantics in Section 3.

## 3 Operational Semantics

Let us now start on our first task: the definition of the operational semantics. As already indicated there are a number of choices to be made. Here we shall just consider three kinds of semantics: a substitution based semantics in the manner of the \( \lambda \)-calculus [1], a structural operational semantics with environments in the manner of [13] and a variant involving explicit alpha-renaming of bound variables (in the manner of the structural congruence used in process algebras like the \( \pi \)-calculus).

All of these semantics are small-step and this is advantageous for the ability to express the correctness of looping programs and for programs with concurrency. The general form of the correctness result then is a subject reduction result:

- If the analysis \( \hat{\rho} \) is acceptable for the expression \( e_1 \)
- and if \( e_1 \) evolves into \( e_2 \)
- then the analysis \( \hat{\rho} \) is acceptable for the expression \( e_2 \).

If a big-step semantics had been used then looping programs\(^\dagger\) as well as concurrent programs would present obstacles to the development.

### Substitution based semantics

Perhaps the simplest kind of operational semantics uses substitutions rather than environments. In this case the operational semantics is given by a judgement of the form

\[ e_1 \xrightarrow{g} e_2 \]

saying that one step of evaluation of \( e_1 \) yields \( e_2 \). To define the judgement it is helpful to clarify that the expressions playing the role of values (i.e. fully evaluated expressions) are simply those function abstractions that only contain global variables:

\[ v \in \text{Val} \]

\[ v ::= \text{fn} \: x :=> \: e \text{ provided that } \text{FV}(\text{fn} \: x :=> \: e) \subseteq \text{FV}(e) \]

Here the set \( \text{FV}(e) \) of free variables of an expression \( e \) is defined in the standard way: \( \text{FV}(x) = \{x\} \), \( \text{FV}(\text{fn} \: x :=> \: e) = \text{FV}(e) \setminus \{x\} \) and \( \text{FV}(e_1 \: e_2) = \text{FV}(e_1) \cup \text{FV}(e_2) \).

\(^\dagger\) Assuming a standard inductive interpretation of big-step semantics.
The semantics is then defined by the following standard axioms and inference rules; intuitively we shall only be interested in evaluating upon expressions whose only free variables are among the global ones, but formally it suffices that this condition holds for the values:

\[
\begin{align*}
  e_1 \xrightarrow{s} e'_1 \\
  e_1 e_2 \xrightarrow{s} e'_1 e_2 \\
  e_2 \xrightarrow{s} e'_2 \\
  v_1 e_2 \xrightarrow{s} v'_1 e'_2 \\
  v_1 v_2 \xrightarrow{s} e[x \mapsto v_2] \text{ if } v_1 = (\text{fn } x \mapsto e)
\end{align*}
\]

Here \(e[x \mapsto v]\) denotes the expression \(e\) with all free occurrences of the variable \(x\) replaced by the value \(v\). Since the formal parameters were assumed to be distinct from the global variables, \(\text{FV}(e)\), no variable capture can take place; hence there is no need for alpha-renaming any formal parameter and therefore the formal parameters continue to be distinct from the global variables. As usual these axioms and rules are to be interpreted inductively, meaning that the judgement is defined by what can be obtained using the axioms and rules and nothing else.

**Environment based semantics**

A somewhat more complex semantics is obtained by using environments instead of substitutions. Following the pattern in [13] we need to extend the syntax of the language in order to define the structural operational semantics. The changes needed for the extended syntax are as follows:

\[
\begin{align*}
  ie \in \text{IExp} \\
  ie ::= x | \text{fn } x \mapsto e | ie \cdot ie | \text{close } (\text{fn } x \mapsto e) \text{ in } \rho | \text{bind } \rho \text{ in } ie \\
  \rho \in \text{Env} \\
  \rho ::= [\ ] | \rho[x \mapsto v] \\
  v \in \text{IVal} \\
  v ::= \text{close } (\text{fn } x \mapsto e) \text{ in } \rho
\end{align*}
\]

The expression \(\text{close } (\text{fn } x \mapsto e) \text{ in } \rho\) encapsulates an unevaluated abstraction \(\text{fn } x \mapsto e\) with an environment \(\rho\) that gives values to all the free variables in \(\text{fn } x \mapsto e\); this construct is needed because we have decided to design a semantics using environments. The expression \(\text{bind } \rho \text{ in } ie\) encapsulates a partly evaluated expression \(ie\) with a local environment \(\rho\); this construct is needed because we have decided to design a small-step semantics. Note that we retain the distinction between unevaluated expressions, \(e \in \text{Exp}\), and partly evaluated (or intermediate) expressions, \(ie \in \text{IExp}\), in order to clarify the precise points where partly evaluated expressions may occur; this will prove helpful when reasoning about the analysis. Finally, (intermediate) values are just closures, and environments are lists of mappings from variables to values: we write \(\rho[x \mapsto v]\) for the environment \(\rho\) augmented such that the variable \(x\)
now maps to the value \( v \); if there is more than one binding for a given variable we always use the rightmost (most recent) one.

The operational semantics is given by a judgement of the form:
\[
\rho \vdash i_e_1 \xrightarrow{\rho} i_e_2
\]
Here \( \rho \) is the environment in which the expression is to be evaluated. It is defined by the following standard axioms and inference rules:
\[
\rho \vdash x \xrightarrow{\rho} v \text{ if } \rho(x) = v
\]
\[
\rho \vdash (\text{fn } x \Rightarrow e) \xrightarrow{\rho} (\text{close (fn } x \Rightarrow e) \text{ in } \rho)
\]
\[
\rho \vdash i_e_1 \xrightarrow{\rho} i_e'_1
\]
\[
\rho \vdash i_e_2 \xrightarrow{\rho} i_e'_2
\]
\[
\rho \vdash v_1 v_2 \xrightarrow{\rho} (\text{bind } \rho_1[x \mapsto v_2] \text{ in } e) \text{ if } v_1 = (\text{close (fn } x \Rightarrow e) \text{ in } \rho_1)
\]
\[
\rho \vdash (\text{bind } \rho_1 \text{ in } i_e_1) \xrightarrow{\rho} (\text{bind } \rho_1 \text{ in } i_e'_1)
\]
\[
\rho \vdash (\text{bind } \rho_1 \text{ in } v_1) \xrightarrow{\rho} v_1
\]
Once again this definition is to be interpreted inductively.

**Explicit alpha-renaming**

Let us finally consider a variation of the environment-based semantics where there is an explicit rule for alpha-renaming the bound variable of an abstraction. This will allow us to illustrate some of the difficulties that will emerge when analysing process calculi like the \( \pi \)-calculus where a structural congruence (containing alpha-renaming) is defined and incorporated into the semantics.

The operational semantics is then given by a judgement of the form:
\[
\rho \vdash i_e_1 \xrightarrow{\rho_\alpha} i_e_2
\]
As above \( \rho \) is the environment in which the expression is to be evaluated. The inductive definition is given by

analogues of the axioms and rules given for \( \xrightarrow{\rho} \)
together with the rule
\[
\frac{ie =_\alpha i'e' \quad \rho \vdash i'e' \xrightarrow{\rho_\alpha} i'e'' \quad i'e'' =_\alpha i'e'''}{\rho \vdash i'e' \xrightarrow{\rho} i'e'''}
\]
where \( ie =_\alpha i'e' \) denotes that \( ie \) is equivalent to \( i'e' \) modulo alpha-renaming (of formal parameters).

A similar modification, \( \xrightarrow{\rho_\alpha} \), is possible for the substitution based semantics; we shall write \( \xrightarrow{\alpha} \) when it is not of any importance whether we refer to \( \xrightarrow{\rho} \) or \( \xrightarrow{\rho_\alpha} \).
4 Flow Logic

Let us now turn to our second task: the specification of the program analysis. As already indicated there are a number of choices to be made. Here we shall concentrate on an abstract specification (in the manner of abstract interpretation [2]), a compositional (or syntax-directed) specification, and a specification using representations of expressions. We shall only be concerned with specifying how to check that a proposed solution is indeed acceptable; the existence of “best” (or least) acceptable solutions is treated in [9,5,11]. Also we shall only deal with succinct specifications as they exhibit a logical flavour that is well suited for semantic considerations.

4.1 Abstract Specification

The most general approach is motivated by the considerations of the collecting semantics (static semantics [2]) in abstract interpretation and works well for open systems. Here the judgement

$$\hat{\rho} \models^A e : W$$

expresses that the pair \((\hat{\rho}, W)\) is an acceptable analysis for the expression \(e\): the \(W\) component describes the set of function abstractions that \(e\) could result in and the \(\hat{\rho}\) component describes the set of function abstractions that the variables inside \(e\) could be bound to. It is defined by the following clauses:

$$\hat{\rho} \models^A x : W \iff \hat{\rho}(x) \subseteq W$$

$$\hat{\rho} \models^A \text{fn } x \to e : W \iff \{\text{fn } x \to e\} \subseteq W$$

$$\hat{\rho} \models^A e_1 \cdot e_2 : W \iff \exists W_1, W_2 : \hat{\rho} \models^A e_1 : W_1 \land \hat{\rho} \models^A e_2 : W_2 \land \forall (\text{fn } x \to e') \in W_1 : \exists W' : W_2 \subseteq \hat{\rho}(x) \land \hat{\rho} \models^A e' : W' \land W' \subseteq W$$

The axiom for variables is straightforward: since the abstract environment records the set of abstract values that the variable can be bound to, we just ensure that this set is part of the set of abstract variables that can result from the expression. Also the axiom for function abstractions is straightforward: a function abstraction gives rise to a single abstract value and we ensure that it is part of what can result from the expression. The rule for function applications is a bit more complex. First we must verify that the analysis is correct as regards the operator part and as regards the operand part. For each possible function being applied, we then “link” the abstract values for the actual parameter to those for the formal parameter, we verify that the analysis is correct as regards the body of the function being called, and finally we “link” the abstract values from the function body into those of the application itself.

This definition is appropriate for open systems because at the function application we analyse the body of the function actually being called rather than assuming that it is part of the program in question and that it has already
been analysed. Hence the functions called can be allowed to come from the environment, e.g. from a library or from the arguments being supplied to the program in question.

*The need for coinduction*

There remains the problem of ensuring that the clauses displayed above do in fact define a relation, $\rho \models e : W$, for each expression $e$. This is complicated by the fact that the definition is not syntax-directed: in the clause for function application we perform an analysis of an expression that need not be a subexpression of the function application in question.

The remedy is standard: we need to interpret the clauses coinductively [9]. To do so we regard the clauses as defining a function

$$S : \mathcal{P}(\text{AE} \times \text{Exp} \times \text{Val}) \rightarrow \mathcal{P}(\text{AE} \times \text{Exp} \times \text{Val})$$

that operates over sets of triples of the form $(\rho, e, W)$ where $\rho \in \text{AE}$, $e \in \text{Exp}$, $W \in \text{Val}$, $\text{AE} = \text{Var} \rightarrow \text{Val}$, $\text{Val} = \mathcal{P}(\text{Exp}^{\infty})$ and $\text{Exp}^{\infty}$ is the set of function abstractions in $\text{Exp}$. The result of $S(S)$ is defined by combining the effect of all the clauses except that any occurrence of $\rho' = e' : W'$ on the right hand side is replaced by $(\rho', e', W') \in S$. We shall omit the detailed definition of $S$ but merely note that the function $S$ is monotonic. By Tarski's Theorem [17] it follows that $S$ has a complete lattice of fixed points. The least fixed point corresponds to the standard inductive interpretation of the clauses whereas the greatest fixed point corresponds to a coinductive definition [8,3].

By taking the coinductive interpretation of the clauses above we obtain the desired definition of $\rho \models e : W$. By reasoning similar to the one in [9] it follows that there always exists a least (or best) analysis that is acceptable in the manner of the above clauses; in particular this means that all programs can be analysed as should not be surprising since one can simply pretend that all function abstractions can reach all places.

*The extended language*

Let us now pause a moment and think ahead. In case the analysis is to be validated with respect to the environment based semantics we will likely have to analyse also the extensions to the syntax. This calls for adding the following clauses

$$\rho \models A \text{ close (fn } x \mapsto e \text{) in } \rho : W \iff \{\text{fn } x \mapsto e\} \subseteq W \land \rho R^A \rho$$

$$\rho \models \text{ bind } \rho \text{ in } \text{i } e : W \iff \exists W' : \rho \models \text{i } e : W' \land W' \subseteq W \land \rho R^A \rho$$

(as well as changing the $e_1, e_2$ above to be $\text{i } e_1 \text{ i } e_2$). The auxiliary relation $R^A$ ensures that the entities encapsulated in the environments have been properly analysed and it is defined by:

$$\rho R^A \rho \iff \forall x \in \text{dom}(\rho) : \forall y_x, e_x, \rho_x :$$
\[ (\rho(x) = \text{close (fn } y_x \Rightarrow e_x \text{) in } \rho_x) \Rightarrow \]
\[ (\{ \text{fn } y_x \Rightarrow e_x \} \subseteq \hat{\rho}(x) \land \rho_x \mathcal{R}^A \hat{\rho}) \]

It is immediate that the auxiliary relation \( \mathcal{R}^A \) is well-defined: in each recursive call the environment gets smaller. The overall collection of clauses is then interpreted coinductively as before.

**Semantic correctness**

Let us now consider the possibility of proving the specification of the analysis correct. Recall that this takes the form of a subject reduction result:

- if the analysis \( \hat{\rho} \) is acceptable for the expression \( e_1 \)
  - and if \( e_1 \) evolves into \( e_2 \)
  - then the analysis \( \hat{\rho} \) is acceptable for the expression \( e_2 \).

(Clearly the detailed formulations depend on the semantics used; they will be spelled out in detail as part of the proofs.)

**Proposition 4.1** The possibility of proving the analysis correct with respect to the semantics is given by the following table:

| \( \models \) | \( \models \) |
|---|---|
| \( \models \) | \( \models \) |
| \( \models s \rightarrow \text{no} \) | \( \models s \rightarrow \text{no} \) |
| \( \models e \rightarrow \text{yes} \) | \( \models e \rightarrow \text{yes} \) |

**Proof.** In each case we must begin with clearly formulating the correctness statement and then either disprove it by means of an example or conduct a formal proof. For the substitution based semantics the subject reduction result reads as follows:

\[ \forall \hat{\rho}, W, e_1, e_2 : (\hat{\rho} \models e_1 : W \land e_1 \xrightarrow{s} e_2) \Rightarrow (\hat{\rho} \models e_2 : W) \]

To disprove this we shall take \( e_1 = (\text{fn } x \Rightarrow (\text{fn } y \Rightarrow (\text{fn } z \Rightarrow x)) (\text{fn } u \Rightarrow u) \), \( e_2 = (\text{fn } y \Rightarrow y) (\text{fn } z \Rightarrow (\text{fn } u \Rightarrow u)) \), \( \hat{\rho}(x) = \{ \text{fn } u \Rightarrow u \}, \hat{\rho}(y) = \{ \text{fn } z \Rightarrow x \}, \hat{\rho}(z) = \emptyset, \hat{\rho}(u) = \emptyset \), and \( W = \{ \text{fn } z \Rightarrow x \} \).

For the environment based semantics the subject reduction result reads as follows:

\[ \forall \hat{\rho}, W, i e_1, i e_2, \rho : (\hat{\rho} \models i e_1 : W \land \rho \mathcal{R}^A \hat{\rho} \land \rho \vdash i e_1 \xrightarrow{e} i e_2) \Rightarrow (\hat{\rho} \models i e_2 : W) \]

We proceed by induction on \( \rho \vdash i e_1 \xrightarrow{e} i e_2 \). In several cases we make use of the lemma

- if \( \hat{\rho} \models e : W_1 \) and \( W_1 \subseteq W_2 \) then \( \hat{\rho} \models e : W_2 \)

that can be proved by inspecting each of the clauses defining \( \models \) in turn.
The negative result for $\longrightarrow$ holds for $\stackrel{\text{sa}}{\longrightarrow}$ as well as $\stackrel{Ea}{\longrightarrow}$ (assuming that the correctness statements are analogues of those displayed above). In the case of $\stackrel{\text{sa}}{\longrightarrow}$ one takes $e_1 = (\text{fn } x \Rightarrow (\text{fn } y \Rightarrow y) (\text{fn } u \Rightarrow u))$, $e_2 = (\text{fn } y \Rightarrow y) (\text{fn } v \Rightarrow v)$, $\hat{\rho}(x) = \{\text{fn } u \Rightarrow u\}$, $\hat{\rho}(y) = \{\text{fn } z \Rightarrow z\}$, $\hat{\rho}(z) = \emptyset$, $\hat{\rho}(u) = \emptyset$, $\hat{\rho}(v) = \emptyset$, and $W = \{\text{fn } z \Rightarrow z\}$. The proof in the case of $\stackrel{Ea}{\longrightarrow}$ is similar.

Remark
Given the lemma in the proof of Proposition 4.1 it might seem that one could simplify the clauses for $\vdash$ by not writing all constraints explicitly in all clauses: one could consider adding a clause saying that $\hat{\rho} = e : W_2$ iff $\exists W_1 : W_1 \subseteq W_2 \land \hat{\rho} = e : W_1$ and then the clause for function abstractions would simply read $\hat{\rho} \vdash \text{fn } x \Rightarrow e : \{\text{fn } x \Rightarrow e\}$ and "similarly" for the other clauses. This is indeed a common trick used in (inductively defined) type systems but unfortunately it turns out to be problematic for the (coinductively defined) clauses here. The reason is that the coinductive interpretation of the revised definition of $\vdash$ yields the relation that is universally true (because of the ability to take $W_1 = W_2$)! So to allow the simplifications (as is done in [11]) one needs a more sophisticated way of interpreting the clauses (and the offending clause must not be added).

4.2 Compositional Specification

In order to obtain a specification that is readily implementable one usually needs to proceed in a more syntax-directed manner. This amounts to checking the bodies of functions when they are "defined" rather than when they are called. One problem with this approach is that we may then end up analysing the bodies of functions that are never called; this can be remedied by adding a reachability component to the analysis (see e.g. [5]) but for conciseness of the presentation we shall abstain from doing so. Another problem with this approach is that we then confine the attention to closed systems; we cannot deal with functions that are not part of the program in question (or some a priori given library or set of arguments to the program).

Analysing function bodies when "defined" then necessitates an additional component to the analysis: a mechanism for ensuring that the set of function abstractions that can result from the body will be known at all relevant application points. Since we have assumed that all function abstractions have initially been alpha-renamed to have distinct formal parameters, it makes sense to use the formal parameter as the unique identifier for the function abstraction in question. We then extend the global analysis information, $\hat{\rho}$, to contain a component $\hat{\rho}(x \Rightarrow) = W$ whenever the body of $\text{fn } x \Rightarrow e$ may yield $W$ (just as $\hat{\rho}(x) = W$ whenever the formal parameter of $\text{fn } x \Rightarrow e$ may be bound to abstract values from $W$).

With these preparations we can then define a judgement of the form
\[ \hat{\rho} \models e : W \]

for expressing that the pair \((\hat{\rho}, W)\) is an acceptable analysis for the expression \(e\) and bearing in mind that the domain of \(\hat{\rho}\) is larger than in the abstract specification. The judgement is defined by the following clauses:

\[
\hat{\rho} \models x : W \quad \text{iff} \quad \hat{\rho}(x) \subseteq W
\]
\[
\hat{\rho} \models \text{fn } x \Rightarrow e : W \quad \text{iff} \quad \{\text{fn } x \Rightarrow e\} \subseteq W \land \hat{\rho} \models e : \hat{\rho}(x \Rightarrow)
\]
\[
\hat{\rho} \models e_1, e_2 : W \quad \text{iff} \quad \exists W_1, W_2 : \hat{\rho} \models e_1 : W_1 \land \hat{\rho} \models e_2 : W_2 \land \\
\forall (\text{fn } x \Rightarrow e') \in W_1 : W_2 \subseteq \hat{\rho}(x) \land \hat{\rho}(x \Rightarrow) \subseteq W
\]

Unlike the abstract specification there is no need to rely on a coinductive definition because the specification is purely compositional (or syntax-directed); however, there is no harm in viewing the specification as being a coinductive definition because the coinductive and inductive definitions turn out to agree (and on philosophical grounds one might indeed argue that one should continue to stress the fact that the specification is coinductive)!

The extended language

Looking ahead to possibly using the environment based semantics for validating the analysis there once more is the need to analyse the extensions to the syntax. This calls for adding the following clauses:

\[
\hat{\rho} \models \text{close } (\text{fn } x \Rightarrow e) \text{ in } \rho : W \quad \text{iff} \quad \{\text{fn } x \Rightarrow e\} \subseteq W \land \rho \overset{c}{\mathcal{R}} \hat{\rho} \land \hat{\rho} \models e : \hat{\rho}(x \Rightarrow)
\]
\[
\hat{\rho} \models \text{bind } \rho \text{ in } i e' : W \quad \text{iff} \quad \exists W' : \hat{\rho} \models i e' : W' \land W' \subseteq W \land \rho \overset{c}{\mathcal{R}} \hat{\rho}
\]

(as well as changing the \(e_1, e_2\) above to be \(ie_1, ie_2\)). The auxiliary relation \(\mathcal{R}_C^c\) is defined as follows:

\[
\rho \overset{c}{\mathcal{R}} \hat{\rho} \quad \text{iff} \quad \forall x \in \text{dom}(\hat{\rho}) : \forall y_x, e_x, \rho_x : \\
(\rho(x) = \text{close } (\text{fn } y_x \Rightarrow e_x) \text{ in } \rho_x) \Rightarrow \\
(\{\text{fn } y_x \Rightarrow e_x\} \subseteq \hat{\rho}(x) \land \rho_x \overset{c}{\mathcal{R}} \hat{\rho} \land \hat{\rho} \models e_x : \hat{\rho}(y_x \Rightarrow))
\]

It is now slightly more tricky to ensure that the analysis and the auxiliary relation are well-defined since they depend recursively upon one another. One possibility is to use mathematical induction on \(n\) to prove that \(\hat{\rho} \models e : W\) and \(\rho \overset{c}{\mathcal{R}} \hat{\rho}\) are well-defined whenever the size of \(e\) and \(\rho\) is at most \(n\); here the size may be taken to be the finite number of ASCII characters needed to represent the entity.

Relationship between the specifications

We said above that the abstract and compositional specifications differ because we did not include a reachability component. Indeed, once this is done along the lines of [5] one can establish an equivalence result between the two
specifications. To give the flavour of this result we state without proof the following weaker fact that holds for the analyses as defined here; it says that (for closed systems) all acceptable analyses with respect to the compositional specification are also acceptable with respect to the abstract specification.

**Fact 4.2** Let \( e_s \) be the given program, let \( \text{Exp}_s \) be the set of subexpressions of \( e_s \) and let \( \text{Exp}_s^{fn} \) be the set of function abstractions in \( e_s \). Assuming that all \( \hat{\rho}(x) \) and \( W \) are restricted to be subsets of \( \text{Exp}_s^{fn} \), i.e. \( \text{AVal} = \mathcal{P}(\text{Exp}_s^{fn}) \), we have

\[
\hat{\rho}^A = e_s : W \iff \hat{\rho}^C = e_s : W.
\]

The opposite implication need not hold; as an example take \( \hat{\rho}(x) = \emptyset, \hat{\rho}(y) = \emptyset, \hat{\rho}(z) = \emptyset, W = \emptyset \) and consider \( e_s = (\text{fn } x \Rightarrow (\text{fn } y \Rightarrow y) (\text{fn } z \Rightarrow z)) \).

**Semantic correctness**

Let us now consider the possibility of establishing a subject reduction result for this analysis.

**Proposition 4.3** The possibility of proving the analysis correct with respect to the semantics is given by the following table:

| \( \sigma \) | \( \rho \) | \( e \) |
|---------------|-----------------|-----------------|
| \( \rightarrow \) | \( \rightarrow^\ast \) | \( \rightarrow^\ast \) |
| \( e \rightarrow^s e_1 \) | \( e \rightarrow^e e_2 \) | \( e \rightarrow^\alpha e_2 \) |
| \( s \) | \( e \) | \( \alpha \) |
| \( \models^C \) | \( \models^C \) | \( \models^C \) |
| no | yes | no |

**Proof.** For the substitution based semantics the subject reduction result reads as follows:

\[
\forall \hat{\rho}, W, e_1, e_2 : (\hat{\rho}^C \models e_1 : W \land e_1 \rightarrow^s e_2) \Rightarrow (\hat{\rho}^C \models e_2 : W)
\]

To disprove this statement we proceed as in the proof of Proposition 4.1.

For the environment based semantics the subject reduction result reads as follows:

\[
\forall \hat{\rho}, W, ie_1, ie_2, \rho : (\hat{\rho}^C \models ie_1 : W \land \rho \mathcal{R}^C \hat{\rho} \land \rho \vdash ie_1 \rightarrow^e ie_2) \Rightarrow (\hat{\rho}^C \models ie_2 : W)
\]

We proceed by induction on \( \rho \vdash ie_1 \rightarrow^e ie_2 \). In several cases we make use of the lemma

if \( \hat{\rho}^C \models e : W_1 \) and \( W_1 \subseteq W_2 \) then \( \hat{\rho}^C \models e : W_2 \)

that can be proved by inspecting each of the clauses defining \( \models^C \) in turn.

The negative result for \( \rightarrow^\ast \) holds for \( \rightarrow^\ast \) as well as \( \rightarrow^s \) (assuming that the correctness statements are analogues of those displayed above); it may be proved as in the proof of Proposition 4.1. \( \Box \)
4.3 Representations of Expressions

In terms of implementing the compositional analysis it would seem that we are carrying a lot of useless baggage around: the complete function bodies. This suggests defining a modified analysis that just uses representations of functions. Given our assumption that all function abstractions in the given program have initially been alpha-renamed so as to have distinct formal parameters, it makes sense to let \( \text{fn } x \) serve as a representation of \( \text{fn } x \Rightarrow e \). We then define the judgement

\[ \hat{\rho} \vdash e : W \]

by the following clauses that are rather directly obtained from those for \( \vdash \):

\[ \hat{\rho} \vdash x : W \iff \hat{\rho}(x) \subseteq W \]

\[ \hat{\rho} \vdash \text{fn } x \Rightarrow e : W \iff \{\text{fn } x\} \subseteq W \land \hat{\rho} \vdash e : \hat{\rho}(x\Rightarrow) \]

\[ \hat{\rho} \vdash e_1, e_2 : W \iff \exists W_1, W_2 : \hat{\rho} \vdash e_1 : W_1 \land \hat{\rho} \vdash e_2 : W_2 \land \forall (\text{fn } x) \in W_1 : W_2 \subseteq \hat{\rho}(x) \land \hat{\rho}(x\Rightarrow) \subseteq W \]

The extended language

The clauses relevant for the extensions to the syntax are minor variations of those considered before:

\[ \hat{\rho} \vdash \text{close } (\text{fn } x \Rightarrow e) \text{ in } \rho : W \iff \{\text{fn } x\} \subseteq W \land \rho \mathcal{R}^{CR} \hat{\rho} \land \hat{\rho} \vdash e : \hat{\rho}(x\Rightarrow) \]

\[ \hat{\rho} \vdash \text{bind } \rho \text{ in } \iota e' : W \iff \exists W' : \hat{\rho} \vdash \iota e' : W' \land W' \subseteq W \land \rho \mathcal{R}^{CR} \hat{\rho} \]

Finally, the definition of the auxiliary relation \( \mathcal{R}^{CR} \) is obtained from the one for \( \mathcal{R}^C \):

\[ \rho \mathcal{R}^{CR} \hat{\rho} \iff \forall x \in \text{dom}(\rho) : \forall y_x, e_x, \rho_x : (\rho(x) = \text{close } (\text{fn } y_x \Rightarrow e_x) \text{ in } \rho_x) \Rightarrow (\{\text{fn } y_x\} \subseteq \hat{\rho}(x) \land \rho_x \mathcal{R}^{CR} \hat{\rho} \land \hat{\rho} \vdash e_x : \hat{\rho}(y_x\Rightarrow)) \]

Well-definedness of these definitions follows as for the compositional specification.

Relationship between the specifications

Intuitively the two compositional specifications should be equivalent because the body of a function abstraction is not used to carry any information. We state without proof the following fact; it states an equivalence result for closed systems.

**Fact 4.4** Let \( e_\ast \) be the given program, let \( \text{Exp}_s \) be the set of subexpressions of \( e_\ast \) and let \( \text{Exp}_s^{\text{fn}} \) be the set of function abstractions in \( e_\ast \). Writing \( \text{ret}(W) = \{ (\text{fn } x) \mid (\text{fn } x \Rightarrow e) \in W \} \) and assuming that all \( \hat{\rho}(x) \) and \( W \) are restricted
to be subsets of $\text{Exp}^l_n$, i.e. $\text{AVal} = \mathcal{P}(\text{Exp}^l_n)$, we have

$$\hat{\rho}^c = e : W \Rightarrow (\text{ret} \circ \hat{\rho})^c \models e : \text{ret}(W)$$

Furthermore, writing $\exp(W) = \{(\text{fn } x \Rightarrow e) \in \text{Exp}_n \mid (\text{fn } x) \in W\}$ and assuming that all $\hat{\rho}(x)$ and $W$ are restricted to be subsets of $\text{ret}^{\text{c}}(\text{Exp}^l_n)$, i.e. $\text{AVal} = \mathcal{P}(\text{ret}(\text{Exp}^l_n))$, we have

$$(\exp \circ \hat{\rho})^c = e : \exp(W) \iff \hat{\rho}^c 
\models e : W$$

Note that $\exp(W)$ produces exactly the same number of elements as in $W$ given our assumption that all function abstractions have initially been alpha-renamed to have distinct formal parameters.

**Semantic correctness**

The use of representations of expressions rather than the expressions themselves turns out to facilitate establishing an analogue of a subject reduction result that defeated us earlier.

**Proposition 4.5** The possibility of proving the analysis correct with respect to the semantics is given by the following table:

| $\models^c$ | $\Rightarrow$ |
|-------------|-------------|
| $s$        | yes         |
| $e$        | yes         |
| $=$        | no          |

**Proof.** For the substitution based semantics the subject reduction reads as follows:

$$\forall \hat{\rho}, W, e_1, e_2 : (\hat{\rho}^c \models e_1 : W \land e_1 \overset{s}{\to} e_2) \Rightarrow (\hat{\rho}^c \models e_2 : W)$$

The proof is by induction on $e_1 \overset{s}{\to} e_2$. For $(\text{fn } x \Rightarrow e) v_2 \overset{s}{\to} e[x \mapsto v_2]$ we use the lemma

if $\hat{\rho}^c \models e : W_1$ and $W_1 \subseteq W_2$ then $\hat{\rho}^c \models e : W_2$

(that can be proved by inspecting each of the clauses defining $\models^c$ in turn) and also the lemma

if $\hat{\rho}^c \models e : W$ and $\hat{\rho}^c \models \hat{\rho}(x)$ then $\hat{\rho}^c \models e[x \mapsto v] : W$

(that can be proved by structural induction on $e$) and we also use the fact that no variable capture can take place in $e[x \mapsto v]$ (because if $v$ is a value then $\text{FV}(v) \subseteq \text{FV}(e_i)$ and the set of formal parameters is disjoint from the set of global variables, $\text{FV}(e_i)$).

For the environment based semantics the subject reduction result reads as follows:
\[ \forall \hat{\rho}, W, i e_1, i e_2, \rho : (\hat{\rho} \models i e_1 : W \land \rho \mathcal{R}^C \hat{\rho} \land \rho \vdash i e_1 \overset{E}{\longrightarrow} i e_2) \Rightarrow (\hat{\rho} \models i e_2 : W) \]

For the proof we proceed as in the proof of the corresponding case in Proposition 4.3.

The negative result for \( \overset{\text{a}}{\longrightarrow} \) holds for \( \overset{\text{s}}{\longrightarrow} \) as well as \( \overset{\text{e}}{\longrightarrow} \) (assuming that the correctness statements are analogues of those displayed above); it may be proved as in the proof of Proposition 4.3.

\[ \square \]

Representations of expressions for abstract specification

The reader might wonder whether it is only in the case of compositional specifications that it is possible to work with representations of expressions rather than the expressions themselves. In the Appendix we shall show that it is indeed possible to do so for abstract specifications although the resulting specification, \( \mathcal{A} \), is not of interest because the analysis is much coarser than the other analyses.

5 Conclusion

Flow Logic is by no means the first approach to formulating program analyses in a logical form. However, in our view it is the first approach that aims at integrating the insights from existing program analysis technologies (such as data flow analysis, control flow analysis and abstract interpretation) into a common form that is applicable to a wide variety of programming languages. This then motivates the current investigation into the relative usefulness of different kinds of semantics.

The technical results concerning the possibility of proving the analyses correct may be summarised as follows:\(^2\):

|         | \( \mathcal{A} \) | \( \mathcal{C} \) | \( \mathcal{CR} \) | \( \mathcal{AR} \) |
|---------|------------------|-----------------|-----------------|------------------|
| \( \overset{\text{s}}{\longrightarrow} \) | no               | no              | yes             | (yes)            |
| \( \overset{\text{E}}{\longrightarrow} \) | yes              | yes             | yes             | (yes)            |
| \( \overset{\text{a}}{\longrightarrow} \) | no               | no              | no              |                  |

Here \( \models \mathcal{A} \) and \( \models \mathcal{CR} \) are equally precise (Fact 4.4) and only \( \text{"slightly coarser"}\) than \( \models \mathcal{C} \) (Fact 4.2) whereas \( \models \mathcal{AR} \) is so coarse as to be of no interest (Fact A.1). The table clearly shows that the environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences\(^3\) (like alpha-renaming) in terms of being able to accomodate a

---

\(^2\) We refer to the Appendix for the missing entry.

\(^3\) Current work on the \( \pi \)-calculus studies techniques aimed at overcoming some of these difficulties; this involves changing the syntax of the language so as to contain "markers" for all entities that are not invariant under the congruence.
variety of specification styles for program analysis.

In our view it is easiest to develop a correct and useful program analysis if one proceeds as follows:

• begin by developing an abstract specification.

This is particularly so for novel calculi involving distribution and mobility because abstract specifications are able to deal with open systems. Also one is less likely to fail to observe that the compositional specifications restrict themselves to closed systems and that a reachability component is needed in order to obtain the same precision as in the abstract specification.

In order to establish semantic correctness by means of a subject reduction result it is a general principle that:

• the analysis information should remain invariant under evaluation.

As we have seen this puts severe demands on the choice of operational semantics: one is more or less forced to abandon working with a simple substitution based semantics in order to work with a more complex environment based semantics; unfortunately, (in keeping with [13]) this requires “artificial” extensions to the syntax, that then also have to be analysed thereby reducing the level of abstraction of the reasoning.

We believe that a compositional (or syntax-directed) specification is a prerequisite for obtaining an efficient implementation. As was explained above, this involves restricting the attention to closed systems. The development in Section 4.2 is semi-compositional [7] in the sense that all expressions considered are subexpressions of the given program; however, semi-compositionality does not suffice for having a free choice between using environment based or substitution based semantics. To achieve this we used representations of expressions in Section 4.3.

In this paper we have only considered succinct specifications and have ignored the verbose formulations of flow logic that are likely to be needed in order to obtain an efficient implementation. One further principle worth stating is that:

• explicit program points (in the form of labelling all subexpressions [9] or demanding all expressions to be in “A-normalform” [4]) are needed for verbose formulations but not for succinct specifications.

For the succinct formulations considered in this paper we merely assumed that all function abstractions had initially been alpha-renamed so as to have distinct formal parameters that were also distinct from the global variables. We refer to [11] for how to transform a succinct specification into a more verbose specification and to [5] for an example of how a verbose specification may be implemented.
Nielson

Acknowledgements

This research was supported in part by the DART-AROS project funded by the Danish Research Councils. We wish to thank Chiara Bodei, Pierpaolo Degano, Chris Hankin, Carolyn Talcott and Mitch Wand for discussions about semantics based program analysis; Torben Amtoft provided helpful comments on a previous version.

References

[1] H. P. Barendregt. The Lambda Calculus — Its Syntax and Semantics, volume 103 of Studies in logic and the foundation of mathematics. North-Holland, revised edition, 1984.

[2] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proc. 4th POPL, pages 238–252. ACM Press, 1977.

[3] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In Proc. POPL ’92, pages 83–94. ACM Press, 1992.

[4] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In Proc. POPL ’95, pages 209–220. ACM Press, 1995.

[5] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In Proceedings of ICFP’97, pages 38–51. ACM Press, 1997.

[6] N. Heintze. Set-based analysis of ML programs. In Proc. LFP’94, pages 306–317, 1994.

[7] N. D. Jones. What not to do when writing an interpreter for specialisation. In Partial Evaluation, pages 216–237. Springer Lecture Notes in Computer Science 1110, 1996.

[8] R. Milner. Communication and Concurrency. Prentice Hall International, 1989.

[9] F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In Proc. POPL ’97, 1997.

[10] F. Nielson, H. R. Nielson, and C. L. Hankin. Principles of Program Analysis: Flows and Effects. To appear in 1999.

[11] H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In Proc. CC’98, to appear in Springer Lecture Notes in Computer Science, 1998.

[12] J. Palsberg. Closure analysis in constraint form. ACM TOPLAS, 17 (1):47–62, 1995.

[13] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
A Representations of Expressions for Abstract Specification

In this appendix we show that it is possible to use representations of expressions also for abstract specifications, although the resulting analysis is so coarse as to be of little interest.

So let us define a judgement

\[ \hat{\rho}^A \models e : W \]

by the following clauses:

\[ \hat{\rho}^A \models x : W \quad \text{iff} \quad \hat{\rho}(x) \subseteq W \]

\[ \hat{\rho}^A \models \text{fn } x \Rightarrow e : W \quad \text{iff} \quad \{\text{fn } x\} \subseteq W \]

\[ \hat{\rho}^A \models e_1 e_2 : W \quad \text{iff} \quad \exists W_1, W_2 : \hat{\rho}^A \models e_1 : W_1 \land \hat{\rho}^A \models e_2 : W_2 \land \forall (\text{fn } x) \in W_1 : \forall \epsilon' : \exists W' : W_2 \subseteq \hat{\rho}(x) \land \hat{\rho}^A \models \epsilon' : W' \land W' \subseteq W \]

For completeness sake we also list the clauses for the extended syntax:

\[ \hat{\rho}^A \models \text{close } (\text{fn } x \Rightarrow e) \text{ in } \rho : W \quad \text{iff} \quad \{\text{fn } x\} \subseteq W \land \rho \mathcal{R}^A \hat{\rho} \]

\[ \hat{\rho}^A \models \text{bind } \rho \text{ in } i \epsilon' : W \quad \text{iff} \quad \exists W' : \hat{\rho}^A \models i \epsilon' : W' \land W' \subseteq W \land \rho \mathcal{R}^A \hat{\rho} \]

The definition of the auxiliary relation then is:

\[ \rho \mathcal{R}^A \hat{\rho} \quad \text{iff} \quad \forall x \in \text{dom}(\rho) : \forall y_x, e_x, \rho_x : \]

\[ (\rho(x) = \text{close } (\text{fn } y_x \Rightarrow e_x) \text{ in } \rho_x ) \Rightarrow (\{\text{fn } y_x\} \subseteq \hat{\rho}(x) \land \rho_x \mathcal{R}^A \hat{\rho}) \]

Relationship between the specifications

The following result shows that the only acceptable analysis for unevaluated programs is the one that says that all function abstractions can reach all places. For the formal statement we need a few preparations. First note that an expression in the original syntax is either an application, a variable or a
function abstraction; if it is an application it can be “maximally expanded” into one of the forms \(((x \, e_1 \, e_2) \, \cdots \, e_n)\) or \(((\text{fn} \, y = e) \, e_1 \, e_2) \, \cdots \, e_n)\) (for \(n > 0\)). To get access to the variable \(x\) occurring to the very left, if indeed such a variable exists, we shall define the set \(\text{LV}(e)\) as follows: \(\text{LV}(x) = \{x\}\), \(\text{LV}(\text{fn} \, x = e) = \emptyset\) and \(\text{LV}(e_1 \, e_2) = \text{LV}(e_1)\). Clearly \(\text{LV}(e)\) contains at most one element.

**Fact A.1** Let \(e_*\) be the given program and suppose that it is an application (i.e. it is not a variable or a function abstraction), that \(\forall x \in \text{LV}(e_*): \hat{\rho}(x) \neq \emptyset\) and that (for all \(x\)) \(\hat{\rho}(x)\) and \(W\) are restricted to be subsets of \(\text{ret}(\text{Exp}^f)\) where \(\text{ret}(W) = \{(\text{fn} \, x) \mid (\text{fn} \, x = e) \in W\}\) and \(\text{Exp}^f\) is the set of function abstractions;

\[
\text{if } \hat{\rho}^A = e_* : W \text{ then } \forall x : \hat{\rho}(x) = \text{ret}(\text{Exp}^f) \text{ and } W = \text{ret}(\text{Exp}^f). 
\]

**Proof** (sketch). The key to the proof is that in the clause for application we can choose \(e' = (\text{fn} \, x_i = x_i) \, e_j\) where \(x_i\) ranges through all variables and \(e_j\) ranges through all function abstractions. For the proof note that since \(e_*\) is an application it can be “maximally expanded”: either into \(((x \, e_1 \, e_2) \, \cdots \, e_n)\) (for \(n > 0\)) in which case we know that \(\hat{\rho}(x) \neq \emptyset\), or else into \(((\text{fn} \, y = e) \, e_1 \, e_2) \, \cdots \, e_n)\) (for \(n > 0\)); in both cases we prove the desired result by induction on \(n\).

**Semantic correctness**

Despite our lack of interest in this analysis let us nonetheless consider the possibility of establishing a subject reduction result; in the table below we put answers in parantheses in order to remind us that the analysis is substantially coarser than those previously considered.

**Proposition A.2** The possibility of proving the analysis correct with respect to the semantics is given by the following table:

\[
\begin{array}{|c|c|}
\hline
\text{AR} & \equiv \\
\hline
S \rightarrow & (\text{yes}) \\
E \rightarrow & (\text{yes}) \\
S\alpha \rightarrow & (\text{yes}) \\
E\alpha \rightarrow & (\text{no}) \\
\hline
\end{array}
\]

**Proof** (sketch). The formulations of the subject reduction results are much as in the proofs of Proposition 4.1; we shall dispense with repeating them here.

We first consider the case of the substitution based semantics. One way to prove the result is to proceed as in the proof of the corresponding case in Proposition 4.5. — A more “abstract” way of proving the result proceeds as follows where we must take care to restrict all \(\hat{\rho}(x)\) and \(W\) to be subsets
of \text{ret}(\text{Exp}^n)$. If an expression (in the original syntax) can evaluate into another then it must be an application and hence it must expand into either \(((x \ e_1) \ e_2) \cdots e_n\) (for \(n > 0\)) or else \(((\text{fn} \ y \Rightarrow e) \ e_1) \ e_2) \cdots e_n\) (for \(n > 0\)). In fact the first case cannot arise because variables are not values. This leaves us with the second case where it follows from Fact A.1 that the \(\hat{\rho}\) and \(W\) in question must state that all function abstractions reach everywhere; but this suffices for analysing an arbitrary expression since all constraints are then vacuously fulfilled.

We next consider the case of the environment based semantics where we proceed as in the proof of the corresponding case in Proposition 4.1.

The positive result for \(\xrightarrow{s_a}\) may be proved as in the “abstract” way of proving \(\xrightarrow{s}\) above: if there is any possibility of using alpha-renaming we know by Fact A.1 that the \(\hat{\rho}\) and \(W\) in question must state that all function abstractions reach everywhere; but then alpha-renaming is not harmful.

The negative result for \(\xrightarrow{e_a}\) may be obtained as follows: let \(\forall x: \hat{\rho}(x) = \emptyset\), \(W = \{\text{fn} \ x \Rightarrow x\}\), \(\rho = [\ ]\), \(ie_1 = \text{fn} \ x \Rightarrow x\) and \(ie_2 = \text{close} (\text{fn} \ y \Rightarrow y)\) in [\].