Math. Struct. in Comp. Science (2003), *vol.* 13, *pp.* 125–156.
DOI: 10.1017/S0960129502003857 Printed in the United Kingdom

Exact flow analysis

CHRISTIAN MOSSIN†

Maconomy A/S, Vordingborggade 18-22, DK-2100 Copenhagen Ø, Denmark
Email: cmo@maconomy.dk

Received 20 September 2001; revised 10 July 2002

We present a type-based flow analysis for simply typed lambda calculus with booleans, data-structures and recursion. The analysis is *exact* in the following sense: if the analysis predicts a redex, there exists a reduction sequence (using standard reduction plus context propagation rules) such that this redex will be reduced. The precision is accomplished using *intersection* typing.

It follows that the analysis is non-elementary recursive – more surprisingly, the analysis is *decidable*. We argue that the specification of such an analysis provides a good starting point for developing new flow analyses and an important benchmark against which other flow analyses can be compared. Furthermore, we believe that the techniques employed for stating and proving exactness are of independent interest: they provide methods for reasoning about the precision of program analyses.

A preliminary version of this paper has previously been published (Mossin 1997b). The present paper extends, elaborates and corrects this previously published abstract.

1. Introduction

Flow analysis of a program aims at approximating at compile-time the flow of values during execution of the program. This includes relating definitions and uses of first-order values (for example, which booleans can be consumed by a given conditional) but also flow of data-structures and higher-order values (for example, which function-closures can be applied at a given application). Hence, the term 'flow analysis' is used here for the natural generalisation of classical value flow analysis to higher order programming languages.

Flow information is directly useful for program transformations such as constant propagation or firstification, and, by interpreting the value flow in an appropriate domain, for many other program analyses. In addition, information about higher-order value flow can allow us to apply first-order program analysis techniques to higher-order languages.

We present a flow analysis for typed, higher-order functional languages which we prove *exact*: the analysis captures exactly the set of potential redexes in the analysed program. This is done by proving that:

† This work was done while at DIKU, University of Copenhagen.

— The result of analysis is *invariant* under β and δ reduction and expansion. The reduction rules are non-standard – they correspond to standard reduction but modified such that redexes are never discarded.
— The analysis correctly predicts that normal forms have no redexes.
— Since non-standard reduction never discards redexes, terms involving 'fix' can never reduce to a value. We prove that the result of analysing a term involving fix is equivalent to the result of analysing a certain finite unfolding of the term. Hence, all redexes predicted by the analysis will be met in a finite number of reduction steps.

As a consequence of a theorem by Statman (Statman 1979) the problem solved is non-elementary recursive, but we show that the analysis is decidable. While the analysis may not be of immediate practical interest, we believe that it provides a fundamental understanding of the nature of flow analysis. Thus, the analysis can be used both as a starting point in the development of practical analyses, and to give an understanding where other analyses lose precision. Finally, the techniques involved in stating and proving exactness are of independent interest: invariance properties under non-standard reduction rules are useful for characterising the precision and imprecision of program analyses.

2. Outline

Section 3 presents the language we will be analysing. Section 4 presents the analysis using an annotated type system. Section 5 gives a syntax directed version of the type system and proves decidability. Section 6 shows the existence of a best result of the analysis.

Section 7 presents a modification of the semantics of our language such that computation is never discarded. Using this non-standard semantics, Section 8 proves the analysis is sound and Section 9 proves that the flow predicted is invariant under expansion. For example, the set of redexes predicted for $(\lambda x.e)@e'$ only contains one more redex (namely the redex itself) than the set of redexes predicted for $e[e'/x]$. Section 9 does not consider 'fix' (treating 'fix' is problematic since never throwing away a computation implies that all expressions involving 'fix' are non-terminating under the non-standard semantics): Section 10 proves that for any fix-expression, there exists a finite unfolding for which the analysis will predict exactly the same redexes. Section 11 proves that the analysis need not predict any redexes for an expression in normal form.

Section 12 summarises the theorems proved, and proves that the analysis is non-elementary recursive. Finally, Section 13 discusses related work and Section 14 gives our conclusions.

3. Language

We analyse simply typed lambda calculus extended with booleans, pairs and recursion. The expression '$\mathrm{let}^l\ (x, y)$ be e in e'' evaluates e to a pair and binds the first component of the pair to x and the second to y[†].

[†] This construct is chosen instead of separate operators for picking the first and second component of the pair, as these operators discard data. This would imply that the non-standard reduction of Section 7 and the completeness results of Section 9 would be less elegant (though no less true).

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Types:

$$t \ ::= \ \text{Bool} \mid t \to t' \mid t \times t'$$

Type Rules:

$$\text{Id}\ \frac{}{A, x : t \vdash x : t} \qquad \text{Bool-intro}\ \frac{}{A \vdash \text{True}^l : \text{Bool}} \qquad \frac{}{A \vdash \text{False}^l : \text{Bool}}$$

$$\text{Bool-elim}\ \frac{A \vdash e : \text{Bool} \quad A \vdash e' : t \quad A \vdash e'' : t}{A \vdash \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : t} \qquad \text{fix}\ \frac{A, x : t \vdash e : t}{A \vdash \text{fix}^l x.e : t}$$

$$\to\text{-intro}\ \frac{A, x : t \vdash e : t'}{A \vdash \lambda^l x : t.e : t \to t'} \qquad \to\text{-elim}\ \frac{A \vdash e : t' \to t \quad A \vdash e' : t'}{A \vdash e @^l e' : t}$$

$$\times\text{-intro}\ \frac{A \vdash e : t \quad A \vdash e' : t'}{A \vdash (e, e')^l : t \times t'} \qquad \times\text{-elim}\ \frac{A \vdash e : t \times t' \quad A, x : t, y : t' \vdash e' : t''}{A \vdash \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : t''}$$

Contexts:

$$C ::= [\,] \mid \lambda^l x : t.C \mid C @^l e \mid e @^l C \mid \text{fix}^l x.C \mid \text{if}^l\ C\ \text{then}\ e'\ \text{else}\ e'' \mid \text{if}^l\ e\ \text{then}\ C\ \text{else}\ e'' \mid \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ C \mid (C, e')^l \mid (e, C)^l \mid \text{let}^l\ (x, y)\ \text{be}\ C\ \text{in}\ e' \mid \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ C$$

Reduction Rules:

$$\begin{array}{lrcll}
(\beta) & (\lambda^{l'} x.e) @^l e' & \longrightarrow & e[e'/x] & \\
(\delta\text{-if}) & \text{if}^l\ \text{True}^{l'}\ \text{then}\ e\ \text{else}\ e' & \longrightarrow & e & \\
 & \text{if}^l\ \text{False}^{l'}\ \text{then}\ e\ \text{else}\ e' & \longrightarrow & e' & \\
(\delta\text{-let-pair}) & \text{let}^l\ (x, y)\ \text{be}\ (e, e')^{l'}\ \text{in}\ e'' & \longrightarrow & e''[e/x][e'/y] & \\
(\delta\text{-fix}) & \text{fix}^l x.e & \longrightarrow & e[\text{fix}^l x.e/x] & \\
(\text{Context}) & C[e] & \longrightarrow & C[e'] & \text{if } e \longrightarrow e'
\end{array}$$

Fig. 1. Language.

We present the language using the type system of Figure 1. We call the set of expression defined by the type system Exp. In order to refer to sub-expression *occurrences*, we assume that terms are *labelled*.

Definition 3.1. For any given expression $e \in \text{Exp}$, let $\mathscr{L}_e$ denote a finite set of labels. Every occurrence of a (sub)expression e' in e is mapped to a label $l \in \mathscr{L}_e$ – we say that l is the label of e'. We use L to denote sets of labels, that is, $L \subseteq \mathscr{L}_e$.

We assume that labelling is preserved under reduction – hence, a label does not identify a single occurrence of a sub-expression, but a set of sub-expressions (intuitively, residuals of the same original sub-expression).

Figure 1 also presents the standard semantics of our language. As usual, we write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. We assume for all expressions that bound and free variables are distinct, and that this property is preserved (by α-conversion) during reduction.

Definition 3.2. Abstractions, booleans and pairs are called *data*, and applications, conditionals and '$\text{let}^l\ (x, y)$ be e in e'' are called *consumers* – thus β, δ-if and δ-let-pair reductions are *data-consumptions*.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Definition 3.3. For an expression e, a *flow relation* Φ is a relation between consumers and data $\Phi \subseteq \mathscr{L}_e \times \mathscr{L}_e$.

Intuitively, (l, l') is in Φ if l is the label of a consumer, l' is the label of data and the consumer can consume the data.

Definition 3.4. Given a reduction sequence $e \longrightarrow^* e'$, the flow relation for the reduction $\mathscr{R}(e \longrightarrow^* e')$ is defined as follows:

$$\begin{aligned}
\mathscr{R}((\lambda^{l'} x.e)@^l e' \longrightarrow e[e'/x]) &= \{(l, l')\} \\
\mathscr{R}(\text{if}^l \text{ True}^{l'} \text{ then } e \text{ else } e' \longrightarrow e) &= \{(l, l')\} \\
\mathscr{R}(\text{if}^l \text{ False}^{l'} \text{ then } e \text{ else } e' \longrightarrow e') &= \{(l, l')\} \\
\mathscr{R}(\text{let}^l (x, y) \text{ be } (e, e')^{l'} \text{ in } e'' \longrightarrow e''[e/x][e'/y]) &= \{(l, l')\} \\
\mathscr{R}(\text{fix}^l x.e \longrightarrow e[\text{fix}^l x.e/x]) &= \{\} \\
\mathscr{R}(C[e] \longrightarrow C[e']) &= \mathscr{R}(e \longrightarrow e') \\
\mathscr{R}(e \longrightarrow e' \longrightarrow^* e'']) &= \mathscr{R}(e \longrightarrow e') \cup \mathscr{R}(e' \longrightarrow^* e'').
\end{aligned}$$

Flow analysis seeks a safe approximation to the possible consumptions during *any* reduction of a term. That is, given e, a flow analysis will compute Φ such that whenever $e \longrightarrow^* e'$, we have $\mathscr{R}(e \longrightarrow^* e') \subseteq \Phi$.

4. Intersection based flow analysis

Intersection types allow us to state more than one property of an expression and use any of the properties at will. Intersection types are more powerful than F2 polymorphism: any polymorphic type can be regarded as an infinite intersection where each component is forced to have the same fixed structure. We use a version of intersection types that includes a subtype ordering. This formulation originates from work by Barendregt, Coppo and Dezani-Ciancaglini (Barendregt *et al.* 1983).

The set of type formation rules presented in Figure 2 defines *properties*. If e is the analysed expression, a property is a standard type where we add a set of labels $L \subseteq \mathscr{L}_e$ (called an *annotation*) to each type constructor. Furthermore, we allow *intersections* of properties. Note that properties are defined in terms of standard types such that $\wedge$ is only allowed on properties with the same underlying standard type.

Definition 4.1. If $\kappa \in \mathscr{K}(t)$, define $|\kappa| = t$. Extending the definition to environments, $|x_1 : \kappa_1, \ldots, x_n : \kappa_n| = x_1 : |\kappa_1|, \ldots, x_n : |\kappa_n|$.

Figure 2 also presents the subtype relation. The first two rules say that anything that has type $\kappa \wedge \kappa'$ can be given type κ or κ'. The third states that if κ is smaller than κ_1 and smaller than κ_2, then it is also smaller than $\kappa_1 \wedge \kappa_2$. The fourth rule states transitivity. The (Bool) rule states that Bool^{L_1} is smaller than Bool^{L_2} if L_1 is a subset of L_2. The (Arrow) and (Product) rules lift the relation to function and pair types in a contra-variant or co-variant way, respectively. Finally, we have three distribution rules for $\rightarrow$ and $\times$ over

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Type Formation:

$$\text{Bool}\ \frac{}{\text{Bool}^L \in \mathscr{K}(\text{Bool})} \qquad \to\ \frac{\kappa \in \mathscr{K}(t) \quad \kappa' \in \mathscr{K}(t')}{\kappa \to^L \kappa' \in \mathscr{K}(t \to t')}$$

$$\times\ \frac{\kappa \in \mathscr{K}(t) \quad \kappa' \in \mathscr{K}(t')}{\kappa \times^L \kappa' \in \mathscr{K}(t \times t')} \qquad \wedge\ \frac{\kappa \in \mathscr{K}(t) \quad \kappa' \in \mathscr{K}(t)}{\kappa \wedge \kappa' \in \mathscr{K}(t)}$$

Subtype Relation:

$$\wedge\text{-E}\ \frac{}{\vdash \kappa \wedge \kappa' \leqslant \kappa} \qquad \frac{}{\vdash \kappa \wedge \kappa' \leqslant \kappa'} \qquad \wedge\ \frac{\vdash \kappa \leqslant \kappa_1 \quad \vdash \kappa \leqslant \kappa_2}{\vdash \kappa \leqslant \kappa_1 \wedge \kappa_2}$$

$$\text{Trans}\ \frac{\vdash \kappa_1 \leqslant \kappa_2 \quad \vdash \kappa_2 \leqslant \kappa_3}{\vdash \kappa_1 \leqslant \kappa_3} \qquad \text{Bool}\ \frac{L_1 \subseteq L_2}{\vdash \text{Bool}^{L_1} \leqslant \text{Bool}^{L_2}}$$

$$\to\ \frac{\vdash \kappa_1 \leqslant \kappa_1' \quad \vdash \kappa_2 \leqslant \kappa_2' \quad L_1 \subseteq L_2}{\vdash \kappa_1' \to^{L_1} \kappa_2 \leqslant \kappa_1 \to^{L_2} \kappa_2'} \qquad \times\ \frac{\vdash \kappa_1 \leqslant \kappa_1' \quad \vdash \kappa_2 \leqslant \kappa_2' \quad L_1 \subseteq L_2}{\vdash \kappa_1 \times^{L_1} \kappa_2 \leqslant \kappa_1' \times^{L_2} \kappa_2'}$$

$$\wedge\text{-arrow}\ \frac{}{\vdash (\kappa_1 \to^L \kappa_2) \wedge (\kappa_1 \to^L \kappa_2') \leqslant \kappa_1 \to^L (\kappa_2 \wedge \kappa_2')}$$

$$\wedge\text{-pair-L}\ \frac{}{\vdash (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1' \times^L \kappa_2) \leqslant (\kappa_1 \wedge \kappa_1') \times^L \kappa_2}$$

$$\wedge\text{-pair-R}\ \frac{}{\vdash (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1 \times^L \kappa_2') \leqslant \kappa_1 \times^L (\kappa_2 \wedge \kappa_2')}$$

Annotated Type Rules:

$$\text{Id}\ \frac{}{A, x : \kappa \vdash x : \kappa} \qquad \to\text{-I}\ \frac{A, x : \kappa \vdash e : \kappa'}{A \vdash \lambda^l x.e : \kappa \to^{\{l\}} \kappa'}$$

$$\to\text{-E}\ \frac{A \vdash e : \kappa' \to^L \kappa \quad A \vdash e' : \kappa'}{A \vdash e @^l e' : \kappa} \qquad \times\text{-I}\ \frac{A \vdash e : \kappa \quad A \vdash e' : \kappa'}{A \vdash (e, e')^l : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{A \vdash e : \kappa \times^L \kappa' \quad A, x : \kappa, y : \kappa' \vdash e' : \kappa''}{A \vdash \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{Bool-I}\ \frac{}{A \vdash \text{True}^l : \text{Bool}^{\{l\}}} \qquad \frac{}{A \vdash \text{False}^l : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{A \vdash e : \text{Bool}^L \quad A \vdash e' : \kappa \quad A \vdash e'' : \kappa}{A \vdash \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa} \qquad \text{fix}\ \frac{A, x : \kappa \vdash e : \kappa}{A \vdash \text{fix}^l x.e : \kappa}$$

$$\text{Sub}\ \frac{A \vdash e : \kappa \quad \vdash \kappa \leqslant \kappa'}{A \vdash e : \kappa'} \qquad \wedge\text{-I}\ \frac{A \vdash e : \kappa \quad A \vdash e : \kappa'}{A \vdash e : \kappa \wedge \kappa'}$$

Fig. 2. Intersection flow analysis.

intersections. Note that these three rules introduce equivalences since the opposite subtypings are derivable.

We define positive and negative occurrences of an annotation L as follows.

Definition 4.2. Assume the syntax tree for a type κ. If the path from the root of the tree to a type constructor c^L (one of Bool, $\times$ or $\to$) follows the argument branch of $\to$ constructors an even (odd) number of times, then L is said to occur positively (negatively) in κ.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Finally, Figure 2 presents the inference rules defining our flow analysis. Rules for data constructing expressions e with label l add $\{l\}$ to the top constructor of the type. This indicates that e can evaluate to one value only: itself. The subtype rule states that if e has type κ and κ' is less precise than κ, then e has type κ'. The final rule states that if e has got type κ and type κ', then it has got type $\kappa \wedge \kappa'$.

It is easy to show that if $A \vdash e : \kappa$, then $|A| \vdash e : |\kappa|$ by the standard type rules.

We remark that the following rule is admissible:

$$\frac{A, x : \kappa_1 \vdash e : \kappa \quad \vdash \kappa_2 \leqslant \kappa_1}{A, x : \kappa_2 \vdash e : \kappa}.$$

We refer to this as the *strengthened assumption* property.

We call a type derivation using the system of Figure 2 a flow derivation.

Definition 4.3. We define a function $\mathscr{F}$ from flow derivations $\mathscr{T}$ for e to flow relations as follows: let $\mathscr{F}(\mathscr{T}) = \Phi$ where $\Phi \subseteq \mathscr{L}_e \times \mathscr{L}_e$ is the least relation such that whenever one of the rules

$$\rightarrow\text{-E}\ \frac{A \vdash e' : \kappa' \rightarrow^L \kappa \quad A \vdash e'' : \kappa'}{A \vdash e' @^l e'' : \kappa}$$

$$\times\text{-E}\ \frac{A \vdash e' : \kappa \times^L \kappa' \quad A, x : \kappa, y : \kappa' \vdash e'' : \kappa''}{A \vdash \text{let}^l\ (x, y)\ \text{be}\ e'\ \text{in}\ e'' : \kappa''}$$

$$\text{Bool-E}\ \frac{A \vdash e' : \text{Bool}^L \quad A \vdash e'' : \kappa \quad A \vdash e''' : \kappa}{A \vdash \text{if}^l\ e'\ \text{then}\ e''\ \text{else}\ e''' : \kappa}$$

is an inference in $\mathscr{T}$, we have $(l, l') \in \Phi$ for all $l' \in L$.

Hence, $\mathscr{F}$ collects the flow information of the derivation and summarises it as a flow relation.

Let $=$ be the equivalence induced by $\leqslant$. For each t, let $\mathscr{K}(t)/=$ denote the equivalence class under $=$. Since $\mathscr{L}_e$ is finite, we have the following proposition.

Proposition 4.1. Given t, e, we have that $\mathscr{K}(t)/=$ is finite.

Definition 4.4. For every expression e and standard type t, define $\bot_t \in \mathscr{K}(t)$ and $\top_t \in \mathscr{K}(t)$ inductively over t as follows:

$\bot_{\text{Bool}} = \text{Bool}^{\{\}}$
$\bot_{t \times t'} = \bot_t \times^{\{\}} \bot_{t'}$
$\bot_{t \rightarrow t'} = \top_t \rightarrow^{\{\}} \bot_{t'}$
$\top_{\text{Bool}} = \text{Bool}^{\mathscr{L}_e}$
$\top_{t \times t'} = \top_t \times^{\mathscr{L}_e} \top_{t'}$
$\top_{t \rightarrow t'} = \bot_t \rightarrow^{\mathscr{L}_e} \top_{t'}$.

The following proposition states that $\bot_t$ and $\top_t$ are the bottom and top elements of $\mathscr{K}(t)$, respectively.

Proposition 4.2. For every expression e, standard type t and $\kappa \in \mathscr{K}(t)$, we have that $\vdash \bot_t \leqslant \kappa$ and $\vdash \kappa \leqslant \top_t$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Proof. The proof is by induction over the structure of κ. □

Definition 4.5. For each t introduce a special constant, which we also call $\bot_t \in \mathrm{Exp}$ of type $\bot_t$.

5. Decidability

We will give a syntax directed version of our inference system. Coppo, Dezani-Ciancaglini and Veneri (Coppo *et al.* 1981) and van Bakel (van Bakel 1995) have used a similar technique for integrating the non-structural rules in the elimination rules.

The first step is to combine the two non-syntax directed rules into one. Define the relation $A \vdash' e : \kappa$ by replacing the (Sub) and ($\wedge$-I) rules by the following rule:

$$\mathrm{Sub}' \ \frac{\forall i \in I : \quad A \vdash' e : \kappa_i \quad \vdash \kappa_i \leqslant \kappa_i'}{A \vdash' e : \bigwedge_{i \in I} \kappa_i'}.$$

Lemma 5.1.

1 If we can deduce $A \vdash e : \kappa$ from $A \vdash e : \kappa_1 \cdots A \vdash e : \kappa_n$ using only rules (Sub) and ($\wedge$-I) then we can deduce the same from the same assumptions using rule (Sub′) only.

2 If

$$\mathrm{Sub}' \ \frac{\forall i \in I : \quad A \vdash' e : \kappa_i \quad \vdash \kappa_i \leqslant \kappa_i'}{A \vdash' e : \bigwedge_{i \in I} \kappa_i'}$$

is the conclusion of a valid derivation, then $A \vdash e : \bigwedge_i \kappa_i'$ can be inferred using rules (Sub) and ($\wedge$-I) from the same assumptions.

Proof. Part (1) is trivial since (Sub) and ($\wedge$-I) are special cases of (Sub′).

If $I = \{1, \ldots, n\}$, then n applications of (Sub) and $n-1$ applications of ($\wedge$-I) suffices for part (2). □

Lemma 5.1 implies that the resulting system is sound and complete with respect to the original system.

Proposition 5.1. $\vdash'$ is sound and complete with respect to $\vdash$:

Soundness: If

$$\frac{\mathscr{T}}{A \vdash' e : \kappa}$$

is a valid derivation, then there exists a valid derivation

$$\frac{\mathscr{T}'}{A \vdash e : \kappa}$$

such that

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash e : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash' e : \kappa}\right).$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Completeness: If

$$\frac{\mathscr{T}}{A \vdash e : \kappa}$$

is a valid derivation, then there exists a valid derivation

$$\frac{\mathscr{T}'}{A \vdash' e : \kappa}$$

such that

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash' e : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e : \kappa}\right).$$

It is trivial to see that we never need two consecutive applications of rule (Sub′).

Definition 5.1. Define type contexts TC as

$$TC ::= [\,] \mid \kappa \times^L TC \mid TC \times^L \kappa \mid \kappa \to^L TC \mid \kappa \wedge TC \mid TC \wedge \kappa.$$

Definition 5.2. Define function $normalise : \mathscr{K}(t) \to \mathscr{K}(t)$ for all t as follows: $normalise(\kappa)$ is the normal form of κ with respect to the following system:

$$\begin{aligned}
\kappa_1 \to^L (\kappa_2 \wedge \kappa_2') &\longrightarrow (\kappa_1 \to^L \kappa_2) \wedge (\kappa_1 \to^L \kappa_2') \\
(\kappa_1 \wedge \kappa_1') \times^L \kappa_2 &\longrightarrow (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1' \times^L \kappa_2) \\
\kappa_1 \times^L (\kappa_2 \wedge \kappa_2') &\longrightarrow (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1 \times^L \kappa_2') \\
TC[\kappa] &\longrightarrow TC[\kappa'] \qquad \text{if } \kappa \longrightarrow \kappa'.
\end{aligned}$$

Note that $\vdash \kappa = normalise(\kappa)$ for all κ. Also, consider the syntax tree for $normalise(\kappa)$: the path from the root to a $\wedge$ node will either not traverse any other nodes than $\wedge$ or the path will traverse a $\to$ node through the argument child (in other words, conjunctions appear at top-level or not in 'Rank 1' position).

Lemma 5.2. If $normalise(\kappa' \to^L \kappa) = \bigwedge_{i \in I} (\kappa' \to^L \kappa_i)$ then $normalise(\kappa) = \bigwedge_{i \in I} \kappa_i$

Proof. We prove the more general property that if

$$\bigwedge_{i \in I} (\kappa'' \to^L \kappa_i) \longrightarrow^* \bigwedge_{j \in J} (\kappa'' \to^L \kappa_j'),$$

then

$$\bigwedge_{i \in I} \kappa_i \longrightarrow^* \bigwedge_{j \in J} \kappa_j'.$$

The proof is by induction on the number of rewrite steps applied. □

A syntax directed version of the inference system is given in Figure 3. We need a version of the property of strengthened assumptions.

Lemma 5.3. If

$$\frac{\mathscr{T}}{A, x : \kappa_1 \vdash^n e : \kappa}$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

$$\text{Id}\ \frac{}{A, x : \kappa \vdash^n x : \kappa} \qquad \to\text{-I}\ \frac{A, x : \kappa \vdash^n e : \kappa'}{A \vdash^n \lambda^l x.e : \kappa \to^{\{l\}} \kappa'}$$

$$\to\text{-E}\ \frac{A \vdash^n e : \kappa' \to^L \kappa \quad \forall i \in I : A \vdash^n e' : \kappa'_i \quad \vdash \bigwedge_{i \in I} \kappa'_i \leqslant \kappa'}{A \vdash^n e @^l e' : \kappa}$$

$$\times\text{-I}\ \frac{A \vdash^n e : \kappa \quad A \vdash^n e' : \kappa'}{A \vdash^n (e, e')^l : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{\forall i \in I : A \vdash^n e : \kappa_i \times^L \kappa'_i \quad A, x : \bigwedge_{i \in I} \kappa_i, y : \bigwedge_{i \in I} \kappa'_i \vdash^n e' : \kappa''}{A \vdash^n \text{let}^l (x, y) \text{ be } e \text{ in } e' : \kappa''}$$

$$\text{Bool-I}\ \frac{}{A \vdash^n \text{True}^l : \text{Bool}^{\{l\}}} \qquad \frac{}{A \vdash^n \text{False}^l : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{A \vdash^n e : \text{Bool}^L \quad A \vdash^n e' : \kappa' \quad A \vdash^n e'' : \kappa'' \quad \vdash \kappa' \leqslant \kappa \quad \vdash \kappa'' \leqslant \kappa}{A \vdash^n \text{if}^l\ e \text{ then } e' \text{ else } e'' : \kappa}$$

$$\text{fix}\ \frac{\forall i : A, x : \kappa \vdash^n e : \kappa_i \quad \vdash \bigwedge_{i \in I} \kappa_i \leqslant \kappa}{A \vdash^n \text{fix}^l x.e : \kappa_j} \quad \text{for any } j \in I$$

Fig. 3. Syntax directed intersection flow analysis.

is a valid derivation and $\vdash \kappa_2 \leqslant \kappa_1$, then $\mathscr{T}'$, κ' exists such that

$$\vdash \kappa' \leqslant \kappa,$$
$$\frac{\mathscr{T}'}{A, x : \kappa_2 \vdash^n e : \kappa'}$$

is a valid derivation and

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A, x : \kappa_2 \vdash^n e : \kappa'}\right) \subseteq \mathscr{F}\left(\frac{\mathscr{T}}{A, x : \kappa_1 \vdash^n e : \kappa}\right).$$

Proof. The proof is by induction over the structure of e. □

The syntax directed system is sound and complete with respect to the original system in the sense given by the following theorem.

Theorem 5.1.

Soundness: If

$$\frac{\mathscr{T}}{A \vdash^n e : \kappa}$$

is a valid derivation, then there exists a valid derivation

$$\frac{\mathscr{T}'}{A' \vdash e : \kappa'}$$

such that

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A' \vdash e : \kappa'}\right) = \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash^n e : \kappa}\right).$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Completeness: If

$$\frac{\mathscr{T}}{A \vdash e : \kappa}$$

is a valid derivation, then there exists a valid derivation

$$\frac{\mathscr{T}'}{A' \vdash^n e : \kappa'}$$

such that

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A' \vdash^n e : \kappa'}\right) \subseteq \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e : \kappa}\right).$$

Proof. We prove the theorem for $\vdash'$ instead of $\vdash$. Then, the theorem follows by Proposition 5.1.

Soundness is a trivial induction since we have have just incorporated the non-syntax directed rules in the syntax directed ones.

For completeness, we prove in addition to the above that:

If

$$\frac{\mathscr{T}}{A \vdash' e : \kappa}$$

is a valid derivation and $\mathit{normalise}(\kappa) = \bigwedge_{i \in I} \kappa_i$, then there exists a family

$$\frac{\mathscr{T}_i}{A \vdash^n e : \kappa_i'} \quad \text{for } i \in I$$

of derivations such that for all i we have $\vdash \kappa_i' \leqslant \kappa_i$ and

$$\mathscr{F}\left(\frac{\mathscr{T}_i}{A \vdash^n e : \kappa_i'}\right) \subseteq \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash' e : \kappa}\right).$$

We prove completeness by induction over the inference tree for

$$\frac{\mathscr{T}}{A \vdash' e : \kappa}.$$

We will just give a few illustrative cases:

(→-I) Assume a derivation

$$\rightarrow\text{-I}\ \frac{\dfrac{\mathscr{T}}{A, x : \kappa \vdash' e : \kappa'}}{A \vdash' \lambda^l x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

By induction, we find a family of derivations for $i \in I$:

$$\frac{\mathscr{T}_i}{A, x : \kappa \vdash^n e : \kappa_i'}$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

such that if $normalise(\kappa') = \bigwedge_{i\in I} \kappa_i$, then $\vdash \kappa'_i \leqslant \kappa_i$ for all $i \in I$. We construct

$$\rightarrow\text{-I}\ \frac{\dfrac{\mathscr{T}_i}{A, x : \kappa \vdash^n e : \kappa'_i}}{A \vdash^n \lambda^l x.e : \kappa \rightarrow^{\{l\}} \kappa'_i}$$

Now we have $normalise(\kappa \rightarrow^{\{l\}} \kappa') = \bigwedge_{i\in I}(\kappa \rightarrow^{\{l\}} \kappa_i)$ and

$$\vdash \kappa \rightarrow^{\{l\}} \kappa'_i \leqslant \kappa \rightarrow^{\{l\}} \kappa_i$$

for all $i \in I$

(→-E) Assume

$$\rightarrow\text{-E}\ \frac{\dfrac{\mathscr{T}}{A \vdash' e : \kappa' \rightarrow^L \kappa} \quad \dfrac{\mathscr{T}'}{A \vdash' e' : \kappa'}}{A \vdash' e@^l e' : \kappa}$$

Further assume

$$\begin{aligned} normalise(\kappa' \rightarrow^L \kappa) &= \textstyle\bigwedge_{i\in I}(\kappa' \rightarrow^L \kappa_i) \\ normalise(\kappa') &= \textstyle\bigwedge_{j\in J} \kappa'_j. \end{aligned}$$

Then, by induction, there exist families of derivations

$$\frac{\mathscr{T}_i}{A \vdash^n e : \kappa'_i \rightarrow^{L_i} \kappa''_i} \quad \text{and} \quad \frac{\mathscr{T}'_j}{A \vdash^n e' : \kappa'''_j}$$

for $i \in I$ and $j \in J$ such that

$$\vdash \kappa'_i \rightarrow^{L_i} \kappa''_i \leqslant \kappa' \rightarrow^L \kappa_i \quad \text{and} \quad \vdash \kappa'''_j \leqslant \kappa'_j.$$

We construct

$$\frac{\dfrac{\mathscr{T}_i}{A \vdash^n e : \kappa'_i \rightarrow^{L_i} \kappa''_i} \quad \forall j \in J : \dfrac{\mathscr{T}'_j}{A \vdash^n e' : \kappa'''_j} \quad \vdash \bigwedge_{j\in J} \kappa'''_j \leqslant \kappa'_i}{A \vdash^n e@^l e' : \kappa''_i}$$

where $\vdash \bigwedge_{j\in J} \kappa'''_j \leqslant \kappa'_i$ follows from $\vdash \kappa'''_j \leqslant \kappa'_j$, $\vdash \bigwedge_{j\in J} \kappa'_j = \kappa'$ and $\vdash \kappa' \leqslant \kappa'_i$. By Lemma 5.2, we have that $normalise(\kappa) = \bigwedge_{i\in I} \kappa_i$. Finally, $\vdash \kappa''_i \leqslant \kappa_i$ and $L_i \subseteq L$.

(Sub′) Assume

$$\frac{\forall i \in I : \quad \dfrac{\mathscr{T}_i}{A \vdash' e : \kappa_i} \quad \vdash \kappa_i \leqslant \kappa'_i}{A \vdash' e : \bigwedge_{i\in I} \kappa'_i}.$$

Let $\bigwedge_{j\in J_i} \kappa_{ij} = normalise(\kappa_i)$ for all $i \in I$. For each $i \in I$ we have, by induction, families of derivations

$$\frac{\mathscr{T}_{ij}}{A \vdash^n e : \kappa'_{ij}}$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

where $\vdash \kappa'_{ij} \leqslant \kappa_{ij}$ for all $i \in I$ and $j \in J_i$. Now, the family indexed by $I \times J$ fulfills the property we wish to prove. □

We can now show that the analysis is decidable as follows: first note that the subtype relation $\vdash \kappa \leqslant \kappa'$ is decidable. Now, given an expression e, the maximal height of the syntax directed inference tree (leaving out the $\vdash \kappa \leqslant \kappa'$ judgments) is bounded by the size of e. With respect to width, the only interesting rules are ($\rightarrow$-E) and (fix) since the number of assumptions in these rules is not fixed:

1 In the ($\rightarrow$-E) rule we have assumptions $A \vdash^n e' : \kappa'_i$, but the number of such assumptions is bounded by the size of $\mathscr{K}(t)/=$, where $t = |\kappa'_i|$.
2 Similarly, in the (fix) rule we have that the number of assumptions $A, x : \kappa \vdash^{\wedge}_n e' : \kappa_i$ is bounded by the size of $\mathscr{K}(t)/=$, where $t = |\kappa_i|$.

Since $(\mathscr{K}(t)/=, \leqslant)$ is finite for each t, we have that for each e we can construct a finite set of trees, such that the set of all valid derivations is a subset. Checking whether each tree is a valid derivation is clearly decidable, so we can compute all valid derivations.

6. Minimality

This section shows that for every expression e there exists a minimal derivation $\mathscr{T}$ such that $\mathscr{F}(\mathscr{T})$ is the minimal flow relation derivable for e. Given the results of the previous section, we are thus able to compute the most precise flow information derivable from the type system. In the following section, we will show that this minimal result is in a certain sense exact.

In this section, we return to the original formulation $\vdash$ of the analysis.

Definition 6.1. A *label vector* is written as $\langle L_1, \ldots, L_n \rangle$, where $L_1, \ldots, L_n$ are sets of labels. Define point-wise set intersection on label vectors by

$$\langle L_1, \ldots, L_n \rangle \cap \langle L'_1, \ldots, L'_n \rangle = \langle L_1 \cap L'_1, \ldots, L_n \cap L'_n \rangle$$

and vector concatenation by

$$\langle L_1, \ldots, L_i \rangle +\!\!+ \langle L_{i+1}, \ldots, L_n \rangle = \langle L_1, \ldots, L_i, L_{i+1}, \ldots, L_n \rangle.$$

Finally, define point-wise subset inclusion:

$$\langle L_1, \ldots, L_n \rangle \subseteq \langle L'_1, \ldots, L'_n \rangle \quad \text{iff} \quad L_1 \subseteq L'_1 \text{ and} \cdots \text{and } L_n \subseteq L'_n.$$

Definition 6.2. The *vectorising* function *vec* maps $\mathscr{K}(t)$ to label vectors and is defined as follows:

$$\begin{aligned} vec(\mathrm{Bool}^L) &= \langle L \rangle \\ vec(\kappa \times^L \kappa') &= vec(\kappa) +\!\!+ \langle L \rangle +\!\!+ vec(\kappa') \\ vec(\kappa \rightarrow^L \kappa') &= vec(\kappa) +\!\!+ \langle L \rangle +\!\!+ vec(\kappa') \\ vec(\kappa \wedge \kappa') &= vec(\kappa) \cap vec(\kappa'). \end{aligned}$$

Definition 6.3. Define an ordering $\preceq$ on properties $\mathscr{K}(t)$ by

$$\kappa \preceq \kappa' \quad \text{iff} \quad vec(\kappa) \subseteq vec(\kappa').$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

We extend the definition of the ordering to judgments by defining

$$(x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash e : \kappa) \preceq (x_1 : \kappa'_1, \ldots, x_n : \kappa'_n \vdash e : \kappa')$$

iff

$$\kappa \preceq \kappa' \quad \text{and} \quad \kappa_i \preceq \kappa'_i \quad \text{for } i \in \{1, \ldots, n\}.$$

Note that the $\preceq$ ordering is co-variant: for example, the least type of $\lambda^{l_1} x.\text{True}^{l_2}$ is $\text{Bool}^{\{\}} \rightarrow^{\{l_1\}} \text{Bool}^{\{l_2\}}$. The ordering is extended to derivations as follows.

Definition 6.4. Define the ordering $\preceq$ on derivations inductively as follows:

1 If one of (Id), ($\rightarrow$-I), ($\rightarrow$-E), ($\times$-I), ($\times$-E), (Bool-I), (Bool-E) and (fix) is the last rule applied in both derivations, then

$$\frac{\mathscr{T}_1 \cdots \mathscr{T}_n}{A \vdash e : \kappa} \preceq \frac{\mathscr{T}'_1 \cdots \mathscr{T}'_n}{A' \vdash e : \kappa'}$$

if

$$(A \vdash e : \kappa) \preceq (A' \vdash e : \kappa') \quad \text{and} \quad \forall i \in \{1, \cdots, n\} : \mathscr{T}_i \preceq \mathscr{T}'_i.$$

2 For any last rule applied in the second derivation

$$\text{Sub}\ \frac{\mathscr{T}}{A \vdash e : \kappa} \preceq \frac{\mathscr{T}'}{A' \vdash e : \kappa'} \quad \text{if} \quad \mathscr{T} \preceq \frac{\mathscr{T}'}{A' \vdash e : \kappa'}.$$

Similarly, for any last rule applied in the first derivation

$$\frac{\mathscr{T}}{A \vdash e : \kappa} \preceq \text{Sub}\ \frac{\mathscr{T}'}{A' \vdash e : \kappa'} \quad \text{if} \quad \frac{\mathscr{T}}{A \vdash e : \kappa} \preceq \mathscr{T}'.$$

3 For any last rule applied in the second derivation

$$\wedge\text{-I}\ \frac{\mathscr{T}_1 \quad \mathscr{T}_2}{A \vdash e : \kappa_1 \wedge \kappa_2} \preceq \frac{\mathscr{T}'}{A' \vdash e : \kappa'}$$

if

$$\mathscr{T}_1 \preceq \frac{\mathscr{T}'}{A' \vdash e : \kappa'} \quad \text{and} \quad \mathscr{T}_2 \preceq \frac{\mathscr{T}'}{A' \vdash e : \kappa'}.$$

Similarly, for any last rule applied in the first derivation

$$\frac{\mathscr{T}}{A \vdash e : \kappa} \preceq \wedge\text{-I}\ \frac{\mathscr{T}'_1 \quad \mathscr{T}'_2}{A' \vdash e : \kappa'_1 \wedge \kappa'_2}$$

if

$$\frac{\mathscr{T}}{A \vdash e : \kappa} \preceq \mathscr{T}'_1 \quad \text{and} \quad \frac{\mathscr{T}}{A \vdash e : \kappa} \preceq \mathscr{T}'_2.$$

The rules of the inductive definition are overlapping, but it is easy to check that the order of applying the rules makes no difference, so the definition is valid.

The following property follows easily from the definitions of $\preceq$ and $\mathscr{F}$.

Lemma 6.1. If $\mathscr{T} \preceq \mathscr{T}'$, then $\mathscr{F}(\mathscr{T}) \subseteq \mathscr{F}(\mathscr{T}')$.

Hence, we find the smallest (most precise) flow relation by finding the smallest derivation under $\preceq$. We shall now prove that a smallest derivation exists.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Definition 6.5. For any t and $\kappa, \kappa' \in \mathscr{K}(t)$, define $\kappa \sqcap \kappa'$ as follows:

$$\begin{aligned}
\mathrm{Bool}^{L_1} \sqcap \mathrm{Bool}^{L_2} &= \mathrm{Bool}^{L_1 \cap L_2} \\
\kappa_1 \times^{L_1} \kappa_1' \sqcap \kappa_2 \times^{L_2} \kappa_2' &= (\kappa_1 \sqcap \kappa_2) \times^{L_1 \cap L_2} (\kappa_1' \sqcap \kappa_2') \\
\kappa_1 \to^{L_1} \kappa_1' \sqcap \kappa_2 \to^{L_2} \kappa_2' &= (\kappa_1 \sqcap \kappa_2) \to^{L_1 \cap L_2} (\kappa_1' \sqcap \kappa_2') \\
(\kappa_1 \wedge \kappa_1') \sqcap \kappa_2 &= (\kappa_1 \sqcap \kappa_2) \wedge (\kappa_1' \sqcap \kappa_2) \\
\kappa_1 \sqcap \kappa_2 &= \kappa_2 \sqcap \kappa_1.
\end{aligned}$$

Lemma 6.2. For any t and $\kappa, \kappa' \in \mathscr{K}(t)$, we have $\kappa \sqcap \kappa' \preceq \kappa$ and $\kappa \sqcap \kappa' \preceq \kappa'$.

Definition 6.6. Define

$$\begin{aligned}
&(x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash e : \kappa) \sqcap (x_1 : \kappa_1', \ldots, x_n : \kappa_n' \vdash e : \kappa') \\
&\qquad = (x_1 : \kappa_1 \sqcap \kappa_1', \ldots, x_n : \kappa_n \sqcap \kappa_n' \vdash e : \kappa \sqcap \kappa').
\end{aligned}$$

Lemma 6.3. Let $\kappa, \kappa' \in \mathscr{K}(t)$ and for all $i \in \{1, \ldots, n\}$: $\kappa_i, \kappa_i' \in \mathscr{K}(t_i)$. Then

$$(x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash e : \kappa) \sqcap (x_1 : \kappa_1', \ldots, x_n : \kappa_n' \vdash e : \kappa') \preceq (x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash e : \kappa)$$

and

$$(x_1 : \kappa_1, \ldots, x_n : \kappa_n \vdash e : \kappa) \sqcap (x_1 : \kappa_1', \ldots, x_n : \kappa_n' \vdash e : \kappa') \preceq (x_1 : \kappa_1', \ldots, x_n : \kappa_n' \vdash e : \kappa').$$

Definition 6.7. The definition of $\sqcap$ is extended to derivations by the following inductive definition:

1 If one of (Id), ($\to$-I), ($\to$-E), ($\times$-I), ($\times$-E), (Bool-I), (Bool-E) and (fix) is the last rule applied in both derivations, then

$$\frac{\mathscr{T}_1 \cdots \mathscr{T}_n}{A \vdash e : \kappa} \sqcap \frac{\mathscr{T}_1' \cdots \mathscr{T}_n'}{A' \vdash e : \kappa'} = \frac{\mathscr{T}_1 \sqcap \mathscr{T}_1' \cdots \mathscr{T}_n \sqcap \mathscr{T}_n'}{(A \vdash e : \kappa) \sqcap (A' \vdash e : \kappa')}.$$

2 If the last rule applied in the first derivation is (Sub), then for any last rule applied in the second derivation

$$\frac{\dfrac{\mathscr{T}}{A \vdash e : \kappa_1} \quad \vdash \kappa_1 \leqslant \kappa_2}{A \vdash e : \kappa_2} \sqcap \frac{\mathscr{T}'}{A' \vdash e : \kappa'} = \frac{\dfrac{\mathscr{T}}{A \vdash e : \kappa_1} \sqcap \dfrac{\mathscr{T}'}{A' \vdash e : \kappa'} \quad \vdash \kappa_1 \sqcap \kappa' \leqslant \kappa_2 \sqcap \kappa'}{(A \vdash e : \kappa_2) \sqcap (A' \vdash e : \kappa')}.$$

3 If the last rule applied in the first derivation is the ($\wedge$-I) rule, then for any last rule applied in the second derivation

$$\frac{\dfrac{\mathscr{T}_1}{A \vdash e : \kappa_1} \quad \dfrac{\mathscr{T}_2}{A \vdash e : \kappa_2}}{A \vdash e : \kappa_1 \wedge \kappa_2} \sqcap \frac{\mathscr{T}'}{A' \vdash e : \kappa'} = \frac{\dfrac{\mathscr{T}_1}{A \vdash e : \kappa_1} \sqcap \dfrac{\mathscr{T}'}{A' \vdash e : \kappa'} \quad \dfrac{\mathscr{T}_2}{A \vdash e : \kappa_2} \sqcap \dfrac{\mathscr{T}'}{A' \vdash e : \kappa'}}{(A \vdash e : \kappa_1 \wedge \kappa_2) \sqcap (A' \vdash e : \kappa')}.$$

4 $\mathscr{T} \sqcap \mathscr{T}' = \mathscr{T}' \sqcap \mathscr{T}$.

In Lemma 6.5, we show that $\mathscr{T} \sqcap \mathscr{T}'$ is a valid derivation. First, we need a simple lemma.

Lemma 6.4. For all κ_1, κ_2 and κ_3, we have $\vdash \kappa_1 \leqslant \kappa_2$ implies $\vdash \kappa_1 \sqcap \kappa_3 \leqslant \kappa_2 \sqcap \kappa_3$.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Proof. The proof follows by induction on the derivation of $\vdash \kappa_1 \leqslant \kappa_2$. □

Lemma 6.5. If $\mathscr{T}$ and $\mathscr{T}'$ are derivations, then so are $\mathscr{T} \sqcap \mathscr{T}'$ and $\mathscr{T} \sqcap \mathscr{T}' \preceq \mathscr{T}$ and $\mathscr{T} \sqcap \mathscr{T}' \preceq \mathscr{T}'$.

Proof. The lemma follows by induction over the sum of the heights of the two derivations. The cases where the last rule of both derivations are syntax directed follow by Lemma 6.3.

If the last rule of one of the derivations is the (Sub) rule, appeal to the induction hypothesis and use Lemma 6.4.

Assume the last rule of one of the derivations is the (∧-I) rule. That is, assume the situation of case (3) in Definition 6.7. By the induction hypothesis,

$$\frac{\mathscr{T}_1}{A \vdash e : \kappa_1} \sqcap \frac{\mathscr{T}'}{A' \vdash e : \kappa'} \quad \text{and} \quad \frac{\mathscr{T}_2}{A \vdash e : \kappa_2} \sqcap \frac{\mathscr{T}'}{A' \vdash e : \kappa'}$$

are derivations, and by Definition 6.7 (all cases), the conclusions of these two must be

$$A \sqcap A' \vdash e : \kappa_1 \sqcap \kappa' \quad \text{and} \quad A \sqcap A' \vdash e : \kappa_2 \sqcap \kappa'.$$

Using the (∧-I) rule, we find $A \sqcap A' \vdash e : (\kappa_1 \sqcap \kappa') \wedge (\kappa_2 \sqcap \kappa')$. By definition of $\sqcap$, this is equivalent to $A \sqcap A' \vdash e : \kappa_1 \wedge \kappa_2 \sqcap \kappa'$, so the right-hand side of case (3) is indeed a valid derivation.

From the induction hypothesis, we also find

$$\frac{\mathscr{T}_1}{A \vdash e : \kappa_1} \sqcap \frac{\mathscr{T}'}{A' \vdash e : \kappa'} \preceq \frac{\mathscr{T}_1}{A \vdash e : \kappa_1} \quad \text{and} \quad \frac{\mathscr{T}_1}{A \vdash e : \kappa_1} \sqcap \frac{\mathscr{T}'}{A' \vdash e : \kappa'} \preceq \frac{\mathscr{T}'}{A' \vdash e : \kappa'}$$

and similarly for $\mathscr{T}_2$. Now the second part of the lemma follows from Definition 6.4. □

Since there is a finite number of derivations for every expression e, the following corollary is an immediate consequence of Lemmas 6.1 and 6.5.

Corollary 6.1. For each e there exists a *minimal* derivation $\mathscr{T}$ under the $\preceq$ ordering. Furthermore, $\mathscr{F}(\mathscr{T})$ is the minimal flow relation derivable for e.

7. Non-standard semantics

In this section, we give a non-standard semantics that exactly characterises the strength of intersection flow analysis. The semantics is a modification of the standard semantics such that if the flow analysis predicts a potential redex, then reduction to normal form under these semantics will indeed cause the redex to be reduced.

Intuitively, the intersection based analysis loses precision whenever a computation is discarded. For example, a derivation for $\text{if}^{l_1}\ \text{True}^{l_2}\ \text{then False}^{l_3}\ \text{else}\ (\lambda^{l_4} x.x)@^{l_5}\text{False}^{l_6}$ will tell us that λ^{l_4} can be applied at application $@^{l_5}$. If we choose to reduce the conditional, we will discard the else-branch and therefore never perform the reduction predicted by the analysis.

We introduce new syntactic constructs to ensure that this never happens. To avoid discarding a computation when 'if' is reduced, we introduce a new construct 'either'. To

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

avoid discarding function arguments that are not used, we introduce a special syntactic construct 'discard'. The type rules for the extensions of the language look as follows:

$$\text{Discard} \quad \frac{A \vdash e' : \kappa' \quad A \vdash e : \kappa}{A \vdash \text{discard } e' \text{ in } e : \kappa}$$

$$\text{Either} \quad \frac{A \vdash e : t \quad A \vdash e' : t}{A \vdash \text{either}_b\ e \text{ or } e' : t} \quad \text{where } b \in \{\text{True}, \text{False}\}.$$

Call the extended set of expressions Exp_s.

The analysis is also imprecise in another way: it can predict redexes that are 'blocked'. For example, for $(\text{let}^{l_1}\ (x_1, x_2)$ be y in $\lambda^{l_2} z.z)@^{l_3}\text{False}^{l_4}$, where y is free, the analysis will predict that $\lambda^{l_2} z.z$ can be applied to False^{l_4}. Under any (type legal) instantiation of y, this will indeed be a redex, so this is not really an error of the analysis. It is, however, important to model this behavior in the non-standard semantics. We therefore introduce *context propagation* rules in the new semantics to ensure that all potential redexes are reduced.

Figure 4 presents the non-standard reduction system. In rules (β), (δ-if) and (δ-let-pair) data labelled l' is consumed by a consumer labelled l. The remaining rules are non-consumptions. Reduction by the rule $C[e] \longrightarrow_s C[e']$ is a consumption (non-consumption) if $e \longrightarrow_s e'$ is a consumption (non-consumption). Function $\mathscr{R}()$ is defined on the extended reduction system as expected. Note that fix is missing in Figure 4 – we will deal with fix in Section 10.

The extended reduction system is confluent, and all reduction paths reduce the same redexes.

Proposition 7.1 (Confluence). If $e \longrightarrow_s^* e_1$ and $e \longrightarrow_s^* e_2$, there exists e' such that $e_1 \longrightarrow_s^* e'$ and $e_2 \longrightarrow_s^* e'$. Furthermore, $\mathscr{R}(e \longrightarrow_s^* e_1 \longrightarrow_s^* e') = \mathscr{R}(e \longrightarrow_s^* e_2 \longrightarrow_s^* e')$.

Proof. The proof is a standard confluence proof. □

It is easy to see that $\longrightarrow_s$ is strongly normalising. Extending the analysis to handle the new constructs is straightforward:

$$\text{Discard} \quad \frac{A \vdash e' : \kappa' \quad A \vdash e : \kappa}{A \vdash \text{discard}^l\ e' \text{ in } e : \kappa}$$

$$\text{Either} \quad \frac{A \vdash e : \kappa \quad A \vdash e' : \kappa}{A \vdash \text{either}^l_b\ e \text{ or } e' : \kappa} \quad \text{where } b \in \{\text{True}^{l'}, \text{False}^{l'}\}.$$

Definition 7.1. Any expression v such that no e exists with $v \longrightarrow_s e$ is called a *value*.

Proposition 7.2. An expression v is a value if and only if it has the following syntax:

$$\begin{aligned} v &::= \text{let}^l\ (x, y) \text{ be } \overline{v} \text{ in } v' \mid \text{if}^l\ \overline{v} \text{ then } v \text{ else } v' \mid \lambda^l x.v \mid (v, v')^l \mid \\ &\quad \text{either}^l_b\ v \text{ or } v' \mid \text{discard}^l\ v \text{ in } v' \mid \text{True}^l \mid \text{False}^l \mid \overline{v} \\ \overline{v} &::= \overline{v}@^l v' \mid x \mid \bot_t. \end{aligned}$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Contexts:

$$C ::= \text{discard}^{l}\ C \text{ in } e' \mid \text{discard}^{l}\ e \text{ in } C \mid \text{either}^{l}_{b}\ C \text{ or } e \mid \text{either}^{l}_{b}\ e \text{ or } C$$

Reduction Rules:

$$\begin{array}{lll}
(\beta) & (\lambda^{l'} x.e)@^{l} e' \longrightarrow_s e[e'/x] & \text{if } x \in FV(e) \\
 & \longrightarrow_s \text{discard}^{l}\ e' \text{ in } e & \text{otherwise} \\
(\delta\text{-if}) & \text{if}^{l}\ \text{True}^{l'}\ \text{then } e \text{ else } e' \longrightarrow_s \text{either}^{l}_{\text{True}^{l'}}\ e \text{ or } e' & \\
 & \text{if}^{l}\ \text{False}^{l'}\ \text{then } e \text{ else } e' \longrightarrow_s \text{either}^{l}_{\text{False}^{l'}}\ e \text{ or } e' & \\
(\delta\text{-let-pair}) & & \\
 & \text{let}^{l}\ (x,y) \text{ be } (e,e')^{l'} \text{ in } e'' \longrightarrow_s e''[e/x][e'/y] & \text{if } x,y \in FV(e'') \\
 & \longrightarrow_s \text{discard}^{l}\ e' \text{ in } e''[e/x] & \text{if } x \in FV(e'') \text{ and } y \notin FV(e'') \\
 & \longrightarrow_s \text{discard}^{l}\ e \text{ in } e''[e'/y] & \text{if } y \in FV(e'') \text{ and } x \notin FV(e'') \\
 & \longrightarrow_s \text{discard}^{l}\ e \text{ in discard}^{l}\ e' \text{ in } e'' & \text{otherwise} \\
(\text{Context}) & C[e] \longrightarrow_s C[e'] & \text{if } e \longrightarrow_s e'
\end{array}$$

Context Propagation Rules:

$$\begin{array}{ll}
(\text{discard}) & (\text{discard}^{l}\ e_1 \text{ in } e_2)@^{l'} e_3 \\
 & \quad \longrightarrow_s \text{discard}^{l}\ e_1 \text{ in } (e_2@^{l'} e_3) \\
 & \text{let}^{l'}\ (x_1,y_1) \text{ be } (\text{discard}^{l}\ e_1 \text{ in } e_2) \text{ in } e_3 \\
 & \quad \longrightarrow_s \text{discard}^{l}\ e_1 \text{ in } (\text{let}^{l'}\ (x_1,y_1) \text{ be } e_2 \text{ in } e_3) \\
 & \text{if}^{l'}\ (\text{discard}^{l}\ e_1 \text{ in } e_2) \text{ then } e_3 \text{ else } e_4 \\
 & \quad \longrightarrow_s \text{discard}^{l}\ e_1 \text{ in } (\text{if}^{l'}\ e_2 \text{ then } e_3 \text{ else } e_4) \\
(\text{pair}) & (\text{let}^{l}\ (x,y) \text{ be } e_1 \text{ in } e_2)@^{l'} e_3 \\
 & \quad \longrightarrow_s \text{let}^{l}\ (x,y) \text{ be } e_1 \text{ in } (e_2@^{l'} e_3) \\
 & \text{let}^{l}\ (x_1,y_1) \text{ be } (\text{let}^{l'}\ (x_2,y_2) \text{ be } e_1 \text{ in } e_2) \text{ in } e_3 \\
 & \quad \longrightarrow_s \text{let}^{l'}\ (x_2,y_2) \text{ be } e_1 \text{ in } (\text{let}^{l}\ (x_1,y_1) \text{ be } e_2 \text{ in } e_3) \\
 & \text{if}^{l'}\ (\text{let}^{l}\ (x,y) \text{ be } e_1 \text{ in } e_2) \text{ then } e_3 \text{ else } e_4 \\
 & \quad \longrightarrow_s \text{let}^{l}\ (x,y) \text{ be } e_1 \text{ in } (\text{if}^{l'}\ e_2 \text{ then } e_3 \text{ else } e_4) \\
(\text{if}) & (\text{if}^{l}\ e_1 \text{ then } e_2 \text{ else } e_3)@^{l'} e_4 \\
 & \quad \longrightarrow_s \text{if}^{l}\ e_1 \text{ then } (e_2@^{l'} e_4) \text{ else } (e_3@^{l'} e_4) \\
 & \text{let}^{l'}\ (x,y) \text{ be } (\text{if}^{l}\ e_2 \text{ then } e_3 \text{ else } e_4) \text{ in } e_1 \\
 & \quad \longrightarrow_s \text{if}^{l}\ e_2 \text{ then } (\text{let}^{l'}\ (x,y) \text{ be } e_3 \text{ in } e_1) \text{ else } (\text{let}^{l'}\ (x,y) \text{ be } e_4 \text{ in } e_1) \\
 & \text{if}^{l}\ (\text{if}^{l'}\ e_1 \text{ then } e_2 \text{ else } e_3) \text{ then } e_4 \text{ else } e_5 \\
 & \quad \longrightarrow_s \text{if}^{l'}\ e_1 \text{ then } (\text{if}^{l}\ e_2 \text{ then } e_4 \text{ else } e_5) \text{ else } (\text{if}^{l}\ e_3 \text{ then } e_4 \text{ else } e_5) \\
(\text{either}) & (\text{either}^{l}_{b}\ e_1 \text{ or } e_2)@^{l'} e_3 \\
 & \quad \longrightarrow_s \text{either}^{l}_{b}\ (e_1@^{l'} e_3) \text{ or } (e_2@^{l'} e_3) \\
 & \text{let}^{l'}\ (x,y) \text{ be } (\text{either}^{l}_{b}\ e_2 \text{ or } e_3) \text{ in } e_1 \\
 & \quad \longrightarrow_s \text{either}^{l}_{b}\ (\text{let}^{l'}\ (x,y) \text{ be } e_2 \text{ in } e_1) \text{ or } (\text{let}^{l'}\ (x,y) \text{ be } e_3 \text{ in } e_1) \\
 & \text{if}^{l'}\ (\text{either}^{l}_{b}\ e_1 \text{ or } e_2) \text{ then } e_3 \text{ else } e_4 \\
 & \quad \longrightarrow_s \text{either}^{l}_{b}\ (\text{if}^{l'}\ e_1 \text{ then } e_3 \text{ else } e_4) \text{ or } (\text{if}^{l'}\ e_2 \text{ then } e_3 \text{ else } e_4)
\end{array}$$

Fig. 4. Extensions for non-standard reduction.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Proof. It is easy to see that if v has the above syntax, then no e exists such that $v \longrightarrow_s e$.

Let e be any expression. We will prove that if no e_0 exists such that $e \longrightarrow_s e_0$, then e has the above syntax. We prove this by induction on the structure of e. First we note that e cannot be fix $x.e$.

True: This is clearly a value.

False: This is clearly a value.

if e then e' else e'': Clearly, e, e' and e'' have to be values for 'if e then e' else e''' to be a value. Furthermore, e cannot be 'True' or 'False' since the conditional would then reduce to 'either$_{\text{True}}$ e' or e''' or 'either$_{\text{False}}$ e' or e''', and it cannot be a discard, a 'let $(x, y)\ldots$,' another conditional or an 'either' expression since then a context propagation rule would be applicable. Finally, it cannot be an abstraction or a pair since it would not be well-typed. The only things left are applications or variables.

either$_b$ e or e': Clearly both e and e' have to be values.

x: This is clearly a value.

$\lambda x.e'$: This is a value if e' is.

$e'@e''$: Clearly e'' has to be a value. Also e' has to be a value. If e' is a pair or a truth value the expression is not well-typed. If e' is an abstraction, e cannot be a value. Similarly, if e' is a discard, a 'let $(x, y)\ldots$', a conditional or an 'either' expression, one of the context propagation rules is applicable. The only options left for e' are a variable or an application.

(e', e''): This is a value if the components are.

let (x, y) be e' in e'': Both e' and e'' have to be values. If e' is a pair, e would not be a value, and, similarly, if it is another pair destructor or a discard, a context propagation rule would be applicable. Since e must be well typed, the only options left are applications and variables. □

Definition 7.2. We define a reduction system $\longrightarrow_c$ on standard terms as $\longrightarrow$ plus the context propagation rules for (pair) and (if).

Note that the context rule for (if) allows strictly more consumptions even for closed terms. For example, applying the rule to

$$(\text{if}^{l_3}\ \text{True}^{l_4}\ \text{then}\ \lambda^{l_1} x.e\ \text{else}\ \lambda^{l_2} y.e')@^{l_5}\text{True}^{l_6}$$

allows the reduction (l_5, l_2).

Proposition 7.3 characterises $\longrightarrow_s$ in terms of $\longrightarrow_c$: every consumption while reducing a term e to a value using $\longrightarrow_s$, is a *potential* consumption while reducing a term e to a value using $\longrightarrow_c$. To prove this, we need a number of definitions and lemmas.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Definition 7.3. Define call-by-value reduction $\longrightarrow_{s,CBV}$ in the extended system by limiting the applicability of the (β) and (δ-let-pair) reduction rules as follows:

$$
\begin{array}{lll}
(\beta) & (\lambda^{l'}x.e)@^{l}v \longrightarrow_{s,CBV} e[v/x] & \text{if } x \in FV(e) \\
 & \longrightarrow_{s,CBV} \text{discard}^{l}\ v \text{ in } e & \text{otherwise} \\
(\delta\text{-let-pair}) & & \\
\text{let}^{l}\ (x,y) \text{ be } (v,v')^{l'} \text{ in } e & \longrightarrow_{s,CBV} e[v/x][v'/y] & \text{if } x,y \in FV(e) \\
 & \longrightarrow_{s,CBV} \text{discard}^{l}\ v' \text{ in } e[v/x] & \text{if } x \in FV(e) \text{ and } y \notin FV(e) \\
 & \longrightarrow_{s,CBV} \text{discard}^{l}\ v \text{ in } e[v'/y] & \text{if } y \in FV(e) \text{ and } x \notin FV(e) \\
 & \longrightarrow_{s,CBV} \text{discard}^{l}\ v \text{ in discard}^{l}\ v' \text{ in } e & \text{otherwise.}
\end{array}
$$

Lemma 7.1. Let $e, v \in \text{Exp}_s$ be given. If $e \longrightarrow_s v$, then there exists a reduction sequence $e \longrightarrow_{s,CBV} v$.

Proof. The proof follows by confluence and strong normalisation. □

Definition 7.4. Define a mapping $\mathscr{S}$ from Exp_s to Exp:

$$
\begin{array}{ll}
\mathscr{S}(\text{discard}^{l}\ e' \text{ in } e) & = \mathscr{S}(e) \\
\mathscr{S}\left(\text{either}^{l}_{\text{True}^{l'}}\ e \text{ or } e'\right) & = \text{if}^{l}\ \text{True}^{l'} \text{ then } \mathscr{S}(e) \text{ else } \mathscr{S}(e') \\
\mathscr{S}\left(\text{either}^{l}_{\text{False}^{l'}}\ e \text{ or } e'\right) & = \text{if}^{l}\ \text{False}^{l'} \text{ then } \mathscr{S}(e) \text{ else } \mathscr{S}(e') \\
\mathscr{S}(\lambda^{l}x.e) & = \lambda^{l}x.\mathscr{S}(e) \\
\mathscr{S}(e@^{l}e') & = \mathscr{S}(e)@^{l}\mathscr{S}(e') \\
\mathscr{S}((e,e')^{l}) & = (\mathscr{S}(e), \mathscr{S}(e'))^{l} \\
\mathscr{S}(\text{let}^{l}\ (x,y) \text{ be } e \text{ in } e') & = \text{let}^{l}\ (x,y) \text{ be } \mathscr{S}(e) \text{ in } \mathscr{S}(e') \\
\mathscr{S}(\text{True}^{l}) & = \text{True}^{l} \\
\mathscr{S}(\text{False}^{l}) & = \text{False}^{l} \\
\mathscr{S}(\text{if}^{l}\ e \text{ then } e' \text{ else } e'') & = \text{if}^{l}\ \mathscr{S}(e) \text{ then } \mathscr{S}(e') \text{ else } \mathscr{S}(e'').
\end{array}
$$

Lemma 7.2. If $e \longrightarrow^{*}_{s,CBV} e'$, then $\mathscr{S}(e) \longrightarrow^{*}_{c} \mathscr{S}(e')$.

Proof. We use induction on the length of $e \longrightarrow^{*}_{s,CBV} e'$. Reductions $e_1 \longrightarrow_{s,CBV} e_2$ are mimicked by $\mathscr{S}(e_1) \longrightarrow_c \mathscr{S}(e_2)$ except (δ-if), which is ignored. □

Lemma 7.3. Let $e, e' \in \text{Exp}_s$ be given such that $e \longrightarrow_s e'$ and $\mathscr{R}(e \longrightarrow_s e') = \{(l,l')\}$. Then there exists $e'' \in \text{Exp}$ such that $\mathscr{S}(e) \longrightarrow e''$ and $\mathscr{R}(\mathscr{S}(e) \longrightarrow e'') = \{(l,l')\}$.

Proof. The proof is by induction on the context and simple case analysis of the consumption involved. □

Proposition 7.3. Let $e \in \text{Exp}$ and $v \in \text{Exp}_s$ be given such that $e \longrightarrow^{*}_{s} v$ and $(l,l') \in \mathscr{R}(e \longrightarrow^{*}_{s} v)$. Then $v' \in \text{Exp}$ exists such that $e \longrightarrow^{*}_{c} v'$ and $(l,l') \in \mathscr{R}(e \longrightarrow^{*}_{c} v')$.

Proof. By Proposition 7.1, any reduction from e to v using $\longrightarrow_s$ will reduce the same set of redexes. Specifically, we can (using Lemma 7.1) choose $e \longrightarrow^{*}_{s,CBV} v$. But then we can use Lemmas 7.2 and 7.3 to construct the reduction sequence using $\longrightarrow_c$. □

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

8. Soundness

We prove soundness for the analysis under the semantics defined by Figure 4 plus the standard rule for 'fix':

$$\mathrm{fix}^l x.e \longrightarrow_s e[\mathrm{fix}^l x.e/x].$$

Soundness is stated as *intensional* subject reduction – we show that not only is the judgment for the whole expression preserved, but the flow computed by the derivation is preserved. We first prove an intensional version of the substitution lemma.

Lemma 8.1 (Substitution lemma). If

$$\frac{\mathscr{T}}{A, x : \kappa' \vdash e : \kappa}$$

and

$$\frac{\mathscr{T}'}{A \vdash e' : \kappa'},$$

then there exists $\mathscr{T}''$ such that:

1 $\dfrac{\mathscr{T}''}{A \vdash e[e'/x] : \kappa}$

2 $\mathscr{F}\left(\dfrac{\mathscr{T}''}{A \vdash e[e'/x] : \kappa}\right) = \mathscr{F}\left(\dfrac{\mathscr{T}}{A, x : \kappa' \vdash e : \kappa}\right) \cup \mathscr{F}\left(\dfrac{\mathscr{T}'}{A \vdash e' : \kappa'}\right).$

Proof. The proof follows by simple induction on the structure of the derivation of $A, x : \kappa' \vdash e : \kappa$. □

Theorem 8.1 (Subject reduction). If

$$\frac{\mathscr{T}}{A \vdash e : \kappa}$$

and $e \longrightarrow_s e'$, then there exists $\mathscr{T}'$ such that:

1 $\dfrac{\mathscr{T}'}{A \vdash e' : \kappa}$

2 If the reduction lets consumer l consume data l', then

$$\mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash e' : \kappa}\right) \cup \{(l, l')\}$$

3 If the reduction is a non-consumption, then

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash e' : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e : \kappa}\right).$$

Proof. The interesting cases follow from Lemma 8.1. □

9. Completeness

This section proves subject expansion for the language without the 'fix' operator.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Subject expansion can only hold if expansion preserves standard types, so this will be an implicit assumption throughout the section. The subject expansion property we prove is *intensional*, that is, the computed flow information is preserved.

Lemma 9.1. Let e be an expression with $n > 0$ occurrences of variable x. If

$$\frac{\mathscr{T}}{A \vdash e[e'/x] : \kappa}$$

then there exists $\kappa_1 \cdots \kappa_n, \mathscr{T}'$ and $\mathscr{T}_1 \cdots \mathscr{T}_n$ such that:

1 $\dfrac{\mathscr{T}_i}{A \vdash e' : \kappa_i}$

2 $\dfrac{\mathscr{T}'}{A, x : \bigwedge\limits_{i \in \{1,\cdots,n\}} \kappa_i \vdash e : \kappa}$

3 $$\mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e[e'/x] : \kappa}\right) = \bigcup_i \mathscr{F}\left(\frac{\mathscr{T}_i}{A \vdash e' : \kappa_i}\right) \cup \mathscr{F}\left(\frac{\mathscr{T}'}{A, x : \bigwedge\limits_{i \in \{1,\cdots,n\}} \kappa_i \vdash e : \kappa}\right).$$

Proof. The proof is by straightforward induction on the derivation $\mathscr{T}$. □

Definition 9.1. If $\mathscr{T}$ is a derivation where x does not occur, we write $\mathscr{T}, x : \kappa$ for the derivation where $x : \kappa$ is added to all environments.

Note that if $\mathscr{T}$ is a valid derivation and x does not occur free in the derivation, then $\mathscr{T}, x : \kappa$ is a valid derivation.

Theorem 9.1 (Subject expansion). If

$$\frac{\mathscr{T}'}{A \vdash e' : \kappa}$$

and $e \longrightarrow_s e'$, then there exists $\mathscr{T}$ such that:

1 $\dfrac{\mathscr{T}}{A \vdash e : \kappa}$

2 If the reduction lets consumer l consume data l', then

$$\mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash e' : \kappa}\right) \cup \{(l, l')\}$$

3 If the reduction is a non-consumption, then

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash e' : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash e : \kappa}\right).$$

Proof. We use case analysis on the definition of $\longrightarrow_s$, that is, the β and δ rules are the base cases and the context rule is the induction step. We will just show some illustrative cases:

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

(β) If x not free in e, the case is simple. Assume x has $n \geqslant 1$ occurrences in e. Then $(\lambda^{l'} x.e)@^l e' \longrightarrow_s e[e'/x]$. By assumption,

$$\frac{\mathscr{T}}{A \vdash e[e'/x] : \kappa}.$$

By Lemma 9.1, we have $\mathscr{T}_i, \kappa_i$ such that

$$\frac{\mathscr{T}_i}{A \vdash e' : \kappa_i} \quad \text{and} \quad \frac{\mathscr{T}'}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa}.$$

Finally,

$$\frac{\dfrac{\dfrac{\mathscr{T}'}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa}}{A \vdash \lambda^{l'} x.e : \left(\bigwedge_i \kappa_i\right) \to^{\{l'\}} \kappa} \qquad \dfrac{\dfrac{\mathscr{T}_1}{A \vdash e' : \kappa_1} \quad \cdots \quad \dfrac{\mathscr{T}_n}{A \vdash e' : \kappa_n}}{A \vdash e' : \bigwedge_i \kappa_i}}{A \vdash (\lambda^{l'} x.e)@^l e' : \kappa}$$

Part (2) follows from Lemma 9.1.

'Context propagation rules': The (discard) and (pair) cases follow from a simple rearrangement of the derivations. The (if) and (either) cases requires the use of $\wedge$ but are relatively straightforward: we give the first case of (if) as illustration.

Assume

$$(\text{if}^l\ e_1 \text{ then } e_2 \text{ else } e_3)@^{l'} e_4 \longrightarrow_s \text{if}^l\ e_1 \text{ then } (e_2@^{l'} e_4) \text{ else } (e_3@^{l'} e_4).$$

Without loss of generality, we can assume

$$\frac{\dfrac{\mathscr{T}_1}{A \vdash e_1 : \text{Bool}^{L_1}} \qquad \mathscr{T}_2 \qquad \mathscr{T}_3}{A \vdash \text{if}^l\ e_1 \text{ then } (e_2@^{l'} e_4) \text{ else } (e_3@^{l'} e_4) : \kappa}$$

where

$$\mathscr{T}_2 = \frac{\dfrac{\dfrac{\mathscr{T}'_2}{A \vdash e_2 : \kappa_4 \to^{L_2} \kappa_2} \qquad \dfrac{\mathscr{T}''_2}{A \vdash e_4 : \kappa_4}}{A \vdash (e_2@^{l'} e_4) : \kappa_2} \qquad \vdash \kappa_2 \leqslant \kappa}{A \vdash (e_2@^{l'} e_4) : \kappa}$$

$$\mathscr{T}_3 = \frac{\dfrac{\dfrac{\mathscr{T}'_3}{A \vdash e_3 : \kappa'_4 \to^{L_3} \kappa_3} \qquad \dfrac{\mathscr{T}''_3}{A \vdash e_4 : \kappa'_4}}{A \vdash (e_3@^{l'} e_4) : \kappa_3} \qquad \vdash \kappa_3 \leqslant \kappa}{A \vdash (e_3@^{l'} e_4) : \kappa}$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

We can now construct

$$\dfrac{\dfrac{\dfrac{\mathscr{T}_1}{A \vdash e_1 : \mathrm{Bool}^{L_1}} \quad \mathscr{T}_4 \quad \mathscr{T}_5}{\vdash \mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 : (\kappa_4 \wedge \kappa_4') \rightarrow^{L_2 \cup L_3} \kappa} \quad \dfrac{\dfrac{\mathscr{T}_2''}{A \vdash e_4 : \kappa_4} \quad \dfrac{\mathscr{T}_3''}{A \vdash e_4 : \kappa_4'}}{A \vdash e_4 : \kappa_4 \wedge \kappa_4'}}{A \vdash (\mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3)@^{l'} e_4 : \kappa}$$

where

$$\mathscr{T}_4 = \dfrac{\dfrac{\mathscr{T}_2'}{A \vdash e_2 : \kappa_4 \rightarrow^{L_2} \kappa_2} \quad \dfrac{\overline{\vdash \kappa_4 \wedge \kappa_4' \leqslant \kappa_4} \quad \vdash \kappa_2 \leqslant \kappa \quad L_2 \subseteq L_2 \cup L_3}{\vdash \kappa_4 \rightarrow^{L_2} \kappa_2 \leqslant (\kappa_4 \wedge \kappa_4') \rightarrow^{L_2 \cup L_3} \kappa}}{A \vdash e_2 : (\kappa_4 \wedge \kappa_4') \rightarrow^{L_2 \cup L_3} \kappa}$$

and

$$\mathscr{T}_5 = \dfrac{\dfrac{\mathscr{T}_3'}{\vdash e_3 : \kappa_4' \rightarrow^{L_3} \kappa_3} \quad \dfrac{\overline{\vdash \kappa_4 \wedge \kappa_4' \leqslant \kappa_4'} \quad \vdash \kappa_3 \leqslant \kappa \quad L_3 \subseteq L_2 \cup L_3}{\vdash \kappa_4' \rightarrow^{L_3} \kappa_3 \leqslant (\kappa_4 \wedge \kappa_4') \rightarrow^{L_2 \cup L_3} \kappa}}{\vdash e_3 : (\kappa_4 \wedge \kappa_4') \rightarrow^{L_2 \cup L_3} \kappa}$$

□

10. Handling 'fix'

In Section 9 we proved that if $e \longrightarrow_s e'$, analysing e will yield exactly one more redex than analysing e' if the reduction is a consumption and exactly the same redexes if it is not. In Section 11 we will prove that analysing values yields the empty set of redexes. These two properties are sufficient to prove that if e reduces to a value, then the analysis will yield exactly the redexes that are actually reduced. Unfortunately, no expression with a 'fix' expression will ever reduce to a value! This section handles this problem.

Definition 10.1. For any expression e of type t, define

$$e^0 = \bot_t$$
$$e^{n+1} = e[e^n/x].$$

Definition 10.2. For any expression e, define

$$e^{(0)} = x$$
$$e^{(n+1)} = e[e^{(n)}/x]$$

with respect to a variable x free in e.

The idea of this section is to show that there exists m such that the analysis is invariant under expansion of the reduction rule $\mathrm{fix}\, x.e \longrightarrow_s e^m$ (throughout this section, we will assume that x occurs in e, since the problem is trivial otherwise). The strategy is as follows:

1 Show that it is no restriction to consider expressions $\mathrm{fix}\, x.e$ with exactly *one* occurrence of x (Lemma 10.1).

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

2 Show subject expansion for $C[\text{fix}\, x.e^{(m)}] \longrightarrow C[e^n]$ (Corollary 10.1).
3 Show subject expansion for $C[\text{fix}\, x.e] \longrightarrow C[\text{fix}\, x.e^{(m)}]$ (Lemma 10.3).

Lemma 10.1. Let e be an expression with $n > 0$ occurrences of x and no occurrences of y. Let

$$\frac{\mathscr{T}}{A \vdash C[e] : \kappa} \quad \text{and} \quad \frac{\mathscr{T}'}{A' \vdash C[(\lambda^{l'} y.e[y/x])@^l x] : \kappa}$$

be minimal derivations. Then

$$\mathscr{F}\left(\frac{\mathscr{T}}{A \vdash C[e] : \kappa}\right) \cup \{l, l'\} = \mathscr{F}\left(\frac{\mathscr{T}'}{A' \vdash C[(\lambda^{l'} y.e[y/x])@^l x] : \kappa}\right).$$

Proof. The proof is trivial from Theorem 9.1. □

Lemma 10.2. Let e and C be given such that e has exactly one occurrence of x. Then there exists m and $n < m$ such that if

$$\frac{\mathscr{T}}{A \vdash C[e^m] : \kappa}$$

is a minimal derivation, then it contains the judgments $A' \vdash e^n : \kappa'$ and $A'' \vdash e^m : \kappa''$ with $A' \mid_{FV(e)} = A'' \mid_{FV(e)}$ and $\kappa' = \kappa''$.

Proof. Clearly, for every y free in e, the underlying standard types of $A'(y)$ and $A''(y)$ are the same. Similarly, the underlying types of κ' and κ'' are the same. By finiteness of $\mathscr{K}(t)/=$, there are only finitely many pairs $(A \mid_{FV(e)}, \kappa)$, and hence there must exist some m where we meet a judgment, that has occurred earlier in the derivation. □

Corollary 10.1. Let m, n be as computed by Lemma 10.2 and

$$\frac{\mathscr{T}}{A \vdash C[e^m] : \kappa}$$

be the minimal derivation. Then there exists

$$\frac{\mathscr{T}'}{A \vdash C\left[\text{fix}\, x.e^{(m-n)}\right] : \kappa}$$

such that

$$\mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash C[\text{fix}\, x.e^{(m-n)}] : \kappa}\right) \subseteq \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash C[e^m] : \kappa}\right).$$

Proof. The proof is immediate from Lemmas 10.2 and 9.1. □

Lemma 10.3. Let $\mathscr{T}$, A, κ, C and e (with exactly one occurrence of x) be given such that

$$\frac{\mathscr{T}}{A \vdash C\left[\text{fix}\, x.e^{(m)}\right] : \kappa}$$

is a derivation. Then there exists $\mathscr{T}'$ such that

$$\frac{\mathscr{T}'}{A \vdash C[\text{fix}\, x.e] : \kappa} \quad \text{and} \quad \mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash C[\text{fix}\, x.e] : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash C\left[\text{fix}\, x.e^{(m)}\right] : \kappa}\right).$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

Proof. We have the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathscr{T}_0}{A, x : \kappa_m \vdash e^{(0)} : \kappa_0}}{A, x : \kappa_m \vdash e^{(1)} : \kappa_1}}{\cdots}}{A, x : \kappa_m \vdash e^{(m-1)} : \kappa_{m-1}}}{A, x : \kappa_m \vdash e^{(m)} : \kappa_m}}{\dfrac{A \vdash \mathrm{fix}\, x.e^{(m)} : \kappa_m}{A \vdash C[\mathrm{fix}\, x.e^{(m)}] : \kappa}}$$

Dismantle this derivation according to Lemma 9.1 into

$$\dfrac{\mathscr{T}_0}{A, x : \kappa_m \vdash e : \kappa_0} \qquad \dfrac{\mathscr{T}_1}{A, x : \kappa_0 \vdash e : \kappa_1} \qquad \cdots \qquad \dfrac{\mathscr{T}_m}{A, x : \kappa_{m-1} \vdash e : \kappa_m}.$$

By the property of strengthened assumptions, we find

$$\dfrac{\mathscr{T}'_0}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa_0} \qquad \dfrac{\mathscr{T}'_1}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa_1} \qquad \cdots \qquad \dfrac{\mathscr{T}'_m}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa_m}$$

(where $\mathscr{T}'_i$ only differs from $\mathscr{T}_i$ in the assumptions for x). Then, by ($\wedge$-I), we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathscr{T}'_0}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa_0} \quad \cdots \quad \dfrac{\mathscr{T}'_m}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa_m}}{A, x : \bigwedge_i \kappa_i \vdash e : \bigwedge_i \kappa_i}}{A \vdash \mathrm{fix}\, x.e : \bigwedge_i \kappa_i} \qquad \vdash \bigwedge_i \kappa_i \leqslant \kappa_m}{\dfrac{A \vdash \mathrm{fix}\, x.e : \kappa_m}{A \vdash C[\mathrm{fix}\, x.e] : \kappa}}$$

Invariance of the computed flow relations then follows by the construction. □

We summarise Corollary 10.1 and Lemma 10.3 as follows (where we use Lemma 10.1 to generalise to an arbitrary number of occurrences of the fix-bound variable).

Theorem 10.1. Let $C[\mathrm{fix}^l x.e]$ be given. Then there exists m such that if

$$\dfrac{\mathscr{T}}{A \vdash C[e^m] : \kappa},$$

there exists $\mathscr{T}'$ such that

$$\dfrac{\mathscr{T}'}{A \vdash C[\mathrm{fix}^l x.e] : \kappa} \quad \text{and} \quad \mathscr{F}\left(\dfrac{\mathscr{T}'}{A \vdash C[\mathrm{fix}^l x.e] : \kappa}\right) \subseteq \mathscr{F}\left(\dfrac{\mathscr{T}}{A \vdash C[e^m] : \kappa}\right)$$

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

If we apply the reduction rule for fix to $C[\text{fix}^l x.e]$ m times, then, by Theorem 8.1, there exists a derivation $\mathscr{T}_1$ for the resulting term such that

$$\mathscr{F}(\mathscr{T}_1) = \mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash C[\text{fix}^l x.e] : \kappa}\right).$$

But from the derivation $\mathscr{T}_1$, we can construct a derivation $\mathscr{T}_2$ for $C[e^m]$ such that $\mathscr{F}(\mathscr{T}_1) = \mathscr{F}(\mathscr{T}_2)$. Hence we have the following theorem.

Theorem 10.2. Let $C[\text{fix}^l x.e]$ be given. Then there exists m such that if

$$\frac{\mathscr{T}}{A \vdash C[e^m] : \kappa} \quad \text{and} \quad \frac{\mathscr{T}'}{A \vdash C[\text{fix}^l x.e] : \kappa}$$

are minimal, then

$$\mathscr{F}\left(\frac{\mathscr{T}}{A \vdash C[e^m] : \kappa}\right) = \mathscr{F}\left(\frac{\mathscr{T}'}{A \vdash C[\text{fix}^l x.e] : \kappa}\right).$$

11. Normal forms

Definition 11.1. We use the notation $A \wedge A'$ for the environment mapping x to $A(x) \wedge A'(x)$ if x is in the domain of both A and A', and to $A(x)$ or $A'(x)$ if x is not in the domain of A' or not in the domain of A, respectively. We extend this to use the abbreviation $\mathscr{T} \wedge A$ for the derivation where A is intersected with all environments in $\mathscr{T}$ (this derivation contains appropriate subsumption steps at variable occurrences).

Clearly, $\mathscr{T} \wedge A$ is a valid derivation if $\mathscr{T}$ is and $\mathscr{F}(\mathscr{T}) = \mathscr{F}(\mathscr{T} \wedge A)$.

Definition 11.2. We say a property $\kappa \in \mathscr{K}$ is *result-empty* iff all positively occurring labels are the empty set $\{\}$. Similarly, κ is called *argument-empty* iff all negatively occurring labels are the empty set $\{\}$.

The following theorem states that the analysis need not predict any redexes for values.

Theorem 11.1. If v is a value, there exists $A, \kappa, \mathscr{T}$ such that

$$\frac{\mathscr{T}}{A \vdash v : \kappa} \quad \text{and} \quad \mathscr{F}\left(\frac{\mathscr{T}}{A \vdash v : \kappa}\right) = \varnothing.$$

Proof. We will show that:

1 For all v not being variables, applications or $\perp$, there exists $A, \kappa, \mathscr{T}$ such that:
   (a) $\dfrac{\mathscr{T}}{A \vdash v : \kappa}$
   (b) $A(x)$ is result-empty for all x
   (c) κ is argument-empty.
2 For all $\overline{v}$ and result-empty κ there exists $A, \mathscr{T}$ such that:
   (a) $\dfrac{\mathscr{T}}{A \vdash \overline{v} : \kappa}$
   (b) $A(x)$ is result-empty for all x.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

It follows from 2(a) that any expression occurring in a 'consumption' context can be given a type where all positively occurring annotations (in particular the top annotation) are the empty set. Hence, we have

$$\mathscr{F}\left(\frac{\mathscr{T}}{A \vdash v : \kappa}\right) = \varnothing.$$

The proof proceeds by induction over the structure of values. We will just show a few illustrative cases:

$\overline{v}@^l v'$: If v' is not a variable or an application, we have, by induction,

$$\frac{\mathscr{T}'}{A' \vdash v' : \kappa'}$$

where A' is result-empty and κ' is argument-empty. If v' is a variable or an application, then the above is true for any result-empty κ' and, in particular, for the argument- and result-empty κ'.

Let any result-empty κ be given. Then $\kappa' \to^{\{\}} \kappa$ is also result-empty. By induction, there is

$$\frac{\mathscr{T}}{A \vdash \overline{v} : \kappa' \to^{\{\}} \kappa}$$

where A is result-empty.

We now construct

$$\frac{\dfrac{\mathscr{T} \wedge A'}{A \wedge A' \vdash \overline{v} : \kappa' \to^{\{\}} \kappa} \qquad \dfrac{\mathscr{T}' \wedge A}{A \wedge A' \vdash v' : \kappa'}}{A \wedge A' \vdash \overline{v}@^l v' : \kappa}$$

where $A \wedge A'$ is result-empty and κ is any given result-empty type.

if^l $\overline{v}$ then v' else v'': By induction we find $A, \mathscr{T}$ such that

$$\frac{\mathscr{T}}{A \vdash \overline{v} : \mathrm{Bool}^{\{\}}}$$

and $A', A'', \kappa', \kappa'', \mathscr{T}', \mathscr{T}''$ such that

$$\frac{\mathscr{T}'}{A' \vdash v' : \kappa'} \quad \text{and} \quad \frac{\mathscr{T}''}{A'' \vdash v'' : \kappa''}$$

where A, A' and A'' are result-empty and κ', κ'' are argument-empty. Let $A''' = A \wedge A' \wedge A''$. Clearly, there exists an argument empty type κ''' such that

$$\frac{\dfrac{\mathscr{T} \wedge A' \wedge A''}{A''' \vdash \overline{v} : \mathrm{Bool}^{\{\}}} \qquad \dfrac{\dfrac{\mathscr{T}' \wedge A \wedge A''}{A''' \vdash v' : \kappa'} \quad \vdash \kappa' \leqslant \kappa'''}{A''' \vdash v' : \kappa'''} \qquad \dfrac{\dfrac{\mathscr{T}'' \wedge A \wedge A'}{A''' \vdash v'' : \kappa''} \quad \vdash \kappa'' \leqslant \kappa'''}{A''' \vdash v'' : \kappa'''}}{A''' \vdash \text{if } \overline{v} \text{ then } v' \text{ else } v'' : \kappa'''}$$

□

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

12. Summarising the results

Sections 9, 10 and 11 prove that the analysis is exact under non-standard reduction. Let e be any expression. Then:

— By applying Theorem 10.1 to every occurrence of 'fix' in e, we have that there exists a fix-free term e'. By Theorem 10.2, the minimal predictable flow of e and e' is identical.
— By using Theorem 11.1 as a base case, and applying Theorem 9.1 inductively, we have that the minimal predictable flow for e' is exactly the redexes met when reducing e' to a value.

If non-standard reduction reduces e to a value v reducing redexes Φ, then, by Proposition 7.3, for any $(l, l') \in \Phi$ there *exists* a reduction sequence using standard reduction extended with context propagation rules for (pair) and (if) such that (l, l') is reduced.

Theorem 12.1 (Exactness). Let e be any expression and let $\mathscr{T}$ be the minimal derivation for e. Then, for any $(l, l') \in \mathscr{F}(\mathscr{T})$, there exists a reduction sequence $e \longrightarrow_c^* e'$ such that $(l, l') \in \mathscr{R}(e \longrightarrow_c^* e')$.

Note that the context-propagation rules only involve extensions to the lambda calculus; the theorem is true for the pure typed lambda calculus using standard β-reduction.

Statman (Statman 1979) shows that decidability of Henkin's formulation of type theory can be reduced to deciding whether simply typed lambda terms beta-reduces to the Church numeral 0.

It follows that deciding whether $e \longrightarrow^* \lambda x.\lambda y.y$ for simply typed pure lambda expressions e of type $(0 \to 0) \to (0 \to 0)$ is non-elementary recursive.

Assume that our analysis is elementary recursive. Then we can find the minimal derivation $\mathscr{T}$ for $e@^{l_1}(\lambda^{l_2} x_1.x_1)@^{l_3} x_2$ in elementary time. By Theorem 12.1, there exists l_4 such that $(l_4, l_2) \in \mathscr{F}(\mathscr{T})$ if and only if there exists a reduction sequence from $e@^{l_1}(\lambda^{l_2} x_1.x_1)@^{l_3} x_2$, where λ^{l_2} is consumed at some application. But this is the case if and only if e reduces to a lambda term different from $\lambda x.\lambda y.y$. By Statman's theorem, our assumption must be wrong, and we can conclude that our analysis is non-elementary recursive.

13. Related work

Intersection Types It is well known that the intersection type discipline gives an exact characterisation of normalising lambda-terms (Coppo *et al.* 1981; Barendregt *et al.* 1983) – the proof of this is along the same lines as our proof of exactness of the analysis for fix-free expressions: prove invariance under reduction and that all normal-forms are typable.

Type-based analysis Type-based analysis has received considerable attention over the last decade. The analyses can be divided into *Church* and *Curry* style.

In *Church style* analysis, the language of properties is defined in terms of the underlying standard types. A special case is *annotated types* where the language of properties is found by adding annotations to type constructors (we do not consider our analysis

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

an instance of annotated types due to the intersection). Examples of Church style analyses are: binding-time analysis (Nielson and Nielson 1988; Henglein and Mossin 1994; Dussart *et al.* 1995), strictness analysis (Kuo and Mishra 1987; Kuo and Mishra 1989; Wright 1991; Jensen 1992; Benton 1992), boxing analysis (Leroy 1992; Henglein and Jørgensen 1994; Jørgensen 1995), totality analysis (Solberg 1994; Solberg 1995) and flow analysis (Heintze 1995; Mossin 1997a; Faxén 1997). Effect systems also belong in this category (Lucassen and Gifford 1988; Talpin and Jouvelot 1994).

In *Curry style* analysis, the analysis makes no use of the underlying type structure. This often makes the analysis applicable to untyped languages but fails to exploit the structural information of the underlying types. Examples are: binding-time analysis (Gomard 1989), dynamic typing (Henglein 1994) and flow analysis (Banerjee 1997; Heintze and McAllester 1997).

Flow analysis Closure analysis (Sestoft 1991) and control flow analysis (Shivers 1991) both address the same problem as the analysis presented in this paper. These analyses, however, only only deal with the flow of higher order values.

Shivers (Shivers 1991) describes a family of analyses *kCFA* for all *k*. It is shown in Mossin (1997a) that, in contrast to popular belief, 0CFA differs from closure analysis: Shivers' family of analyses is *evaluation-order* dependent[†].

Palsberg later gave a constraint formulation of closure analysis (Palsberg 1994). Palsberg and O'Keefe showed that the type information derived using closure analysis corresponds to Amadio–Cardelli typing (Palsberg and O'Keefe 1995). Independently, Heintze showed the same result and the converse (Heintze 1995), by annotating Amadio–Cardelli typings, closure analysis is derived.

Heintze and McAllester define a variant of Palsberg's constraint formulation of closure analysis (Heintze and McAllester 1997). Termination of the analysis relies on the fact that the analysed program is well-typed – if the size of types is bounded the analysis will run in quadratic time (in contrast to Palsberg's cubic algorithm). The present author independently achieved the same time complexity (Mossin 1998). This variant explicitly uses the structure of the type derivation in the construction of a flow graph.

Shivers' analyses are abstractions of an instrumented semantics. This semantics is defined using *contours* that keep track of different live instances of variables – *k*CFA makes the set of contours isomorphic to $Call^k$ where *Call* is the set of call-sites. The instrumented semantics can be thought of as an exact (though undecidable) flow analysis.

More refined instrumented semantics have been proposed by Jaganathan and Weeks (Jaganathan and Weeks 1995) and by Nielson and Nielson (Nielson and Nielson 1997). Both allow a wider choice of abstractions and thus provide good starting points for the development of practical analyses.

Independently of the present work, Banerjee has studied rank 2 intersection based flow analysis (Banerjee 1997). The goal of his work is practical (modularity and increased precision compared to closure analysis), but an assessment of the practicality (in particular,

[†] The notion of evaluation-order dependency is well known in the imperative data-flow community, but has received little attention for higher-order programming languages.

https://doi.org/10.1017/S0960129502003857 Published online by Cambridge University Press

complexity) of his analysis is made difficult by the lack of a clear-cut separation between standard type system and flow information. Rank 2 intersection is indeed a promising way of abstracting our analysis, but further studies are needed – it is likely that the techniques employed here could be used to prove exactness of rank 2 intersection flow analysis on first-order terms.

The present author has described (Mossin 1997a) a family of flow analyses based on types: simple flow analysis, subtype flow analysis (which is equivalent to closure analysis), polymorphic flow analysis (with polymorphic recursion) and the intersection flow analysis presented here. Independently, Faxén gives a flow analysis with let-polymorphism in annotations (Faxén 1997).

14. Conclusion

We have presented a flow analysis for a higher-order typed language with recursion, and proved that the analysis is exact: if the analysis predicts a redex, then there exists a reduction sequence such that this redex will be reduced.

The analysis is decidable but the precision of the analysis implies that it is non-elementary recursive. This is, however, no worse (or better) than the complexity of strictness analysis, and we believe that intersection based flow analysis (as strictness analysis) provides a good starting point for developing practical analyses.

Finally, we believe that the technique of using invariance under reduction as a characterisation of the precision of analyses can be useful for reasoning about other analyses as well. In particular, the idea of proving invariance under non-standard reduction and relating this to standard reduction seems useful.

References

Banerjee, A. (1997) A modular, polyvariant, and type-based closure analysis. In: International Conference on Functional Programming (ICFP'97) Amsterdam, the Netherlands. *SIGPLAN Notices* **32** (8) 1–10.

Barendregt, H., Coppo, M. and Dezani-Ciancaglini, M. (1983) A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic* **48** 931–940.

Benton, N. (1992) *Strictness analysis of lazy functional programs*, Ph. D. thesis, University of Cambridge.

Coppo, M., Dezani-Ciancaglini, M. and Venneri, B. (1981) Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* **27** 45–58.

Dussart, D., Henglein, F. and Mossin, C. (1995) Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In: Mycroft, A. (ed.) Proc. 2nd Int'l Static Analysis Symposium (SAS'95), Glasgow, Scotland. *Springer-Verlag Lecture Notes in Computer Science* **983** 118–135.

Faxén, K.-F. (1997) *Analysing, Transforming and Compiling Lazy Functional Programs*, Ph. D. thesis, Royal Institute of Technology, Sweden.

Gomard, C. (1989) Higher order partial evaluation – hope for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark.

Heintze, N. (1995) Control-flow analysis and type systems. In: Mycroft, A. (ed.) Proc. 2nd Int'l Static Analysis Symposium (SAS'95), Glasgow, Scotland. *Springer-Verlag Lecture Notes in Computer Science* **983** 189–206.

Heintze, N. and McAllester, D. (1997) Linear-time Subtransitive Control Flow Analysis. In: Programming Language Design and Implementation (PLDI'97). *SIGPLAN Notices* **32** (5) 261–272.

Henglein, F. (1994) Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)* **22** (3) 197–230.

Henglein, F. and Jørgensen, J. (1994) Formally optimal boxing. In: *Proceedings of 21st ACM Symposium on Principles of Programming Languages (POPL'94), Oregon* 213–226.

Henglein, F. and Mossin, C. (1994) Polymorphic binding-time analysis. In: Sannella, D. (ed.) Proceedings of European Symposium on Programming. *Springer-Verlag Lecture Notes in Computer Science* **788** 287–301.

Jaganathan, S. and Weeks, S. (1995) A unified treatment of flow analysis in higher-order languages. In: *Principles of Programming Languages (POPL'95)*, ACM Press 393–407.

Jensen, T. (1992) *Abstract Interpretation in Logical Form*, Ph. D. thesis, Imperial College, Univ. of London. (Available as DIKU Report 93/11.)

Jørgensen, J. (1995) *A calculus for boxing analysis of polymorphically typed languages*, Ph. D. thesis, DIKU, University of Copenhagen.

Kuo, T. and Mishra, P. (1987) On strictness and its analysis. In: *Proc. 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87)* 144–155.

Kuo, T. and Mishra, P. (1989) Strictness analysis: A new perspective based on type inference. In: *Proc. Functional Programming Languages and Computer Architecture (FPCA'89), London, England*, ACM Press 260–272.

Leroy, X. (1992) Unboxed objects and polymorphic typing. In: *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'92), Albuquerque, New Mexico*, ACM Press 177–188.

Lucassen, J. M. and Gifford, D. K. (1988) Polymorphic effect systems. In *Principles of Programming Languages (POPL'88), San Diego*, ACM Press 47–57.

Mossin, C. (1997a) *Flow Analysis of Typed Higher-Order Programs*, Ph. D. thesis, DIKU, University of Copenhagen.

Mossin, C. (1997b) Exact Flow Analysis. In: *Static Analysis Symposium (SAS'97)*, Springer-Verlag 250–264.

Mossin, C. (1998) Higher-order value flow graphs. *Nordic Journal of Computing* **5** (3) 214–234.

Nielson, F. and Nielson, H. R. (1997) Infinitary control flow analysis: a collecting semantics for closure analysis. In: *24th Symposium on Principles of Programming Languages (POPL'97)* 332–345.

Nielson, H. R. and Nielson, F. (1988) Automatic binding time analysis for a typed lambda-calculus. In: *Fifteenth ACM Symposium on Principles of Programming Languages (POPL'88)*, ACM Press 98–106. (Extended Abstract.)

Palsberg, J. (1994) Global program analysis in constraint form. In: Tison, S. (ed.) 19th International Colloquium on Trees in Algebra and Programming (CAAP'94). *Springer-Verlag Lecture Notes in Computer Science* **787** 276–290.

Palsberg, J. and O'Keefe, P. (1995) A type system equivalent to flow analysis. In: *Principles of Programming Languages (POPL'95), San Francisco* 367–378.

Sestoft, P. (1991) *Analysis and Efficient Implementation of Functional Languages*, Ph. D. thesis, DIKU, University of Copenhagen.

Shivers, O. (1991) *Control-Flow Analysis of Higher-Order Languages*, Ph. D. thesis, Carnegie Mellon University. (CMU-CS-91-145.)

Solberg, K. L. (1994) Strictness and totality analysis In: Symposium on Static Analysis (SAS'94). *Springer-Verlag Lecture Notes in Computer Science* **864** 408–422.

Solberg, K. L. (1995) Strictness and totality analysis. In: Theory and Practice of Software Development (TAPSOFT'95). *Springer-Verlag Lecture Notes in Computer Science* **915** 501–515.

Statman, R. (1979) The typed λ-calculus is not elementary recursive. *Theoretical Computer Science* **9** (1) 73–81.

Talpin, J.-P. and Jouvelot, P. (1994) The type and effect discipline. *Information and Computation* **111** 245–296.

van Bakel, S. (1995) Intersection type assignment systems. *Theoretical Computer Science* **151** (2) 385–435.

Wright, D. (1991) A new technique for strictness analysis. In: Theory and Practice of Software Development (TAPSOFT'91). *Springer-Verlag Lecture Notes in Computer Science* **494** 235–258.