Exact Flow Analysis

Christian Mossin

DIKU, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark,
e-mail: mossin@diku.dk

Abstract. 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, then 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.

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 (eg. which booleans can be consumed by a given conditional) but also flow of data-structures and higher-order values (eg. which function-closures can be applied at a given application). Values are abstracted by a label of the occurrence of the value — i.e. the label of first-order values uniquely identifies the occurrence of the value, while data-structures and closures are abstracted by the label of the constructor resp. lambda. Thus, closure analysis [26,24] and control flow analysis [27] are special cases focusing on closures.

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. Furthermore, information about higher-order value flow can allow first-order program analysis techniques to be applicable 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

- 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 has no redexes.

– Under non-standard reduction, all terms involving 'fix' are non-terminating. 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 [30] 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 gives 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 understand where other analyses lose information. Finally, the techniques involved in stating and proving exactness are of independent interest: invariance properties under non-standard reduction rules are useful to characterise the precision and imprecision of program analyses.

Outline Section 2 presents the language we will be analysing and section 3 presents the analysis using an annotated typed system. Section 5 shows the existence of a best result of the analysis.

Section 6 presents a modification of the semantics of our language such that computation is never discarded. Using this non-standard semantics, section 7 proves the analysis sound and section 8 proves that the flow predicted is invariant under expansion. E.g. 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 8 does not consider 'fix'(these are problematic since never throwing away computation implies that all expressions involving 'fix' are non-terminating under the non-standard semantics): section 9 proves that for any fix-expression, there exists a finite unfolding for which the analysis will predict exactly the same redexes. Section 10 proves that the analysis need not predict any redexes for expression in normal form.

Section 11 summarises the proven theorems and shows why they form a proof of exactness. Finally, section 12 discusses related work and section 13 concludes.

2 Language

We analyse simply typed lambda calculus extended with booleans, pairs and recursion. The types of the language are

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

We present the language using the type system of figure 1. In order to refer to subexpression *occurrences*, we assume that terms are *labelled*. For any given expression e, the finite set of labels in e is referred to as $\mathcal{L}_e$. 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 subexpressions (intuitively redexes of the same original subexpression).

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:

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

Fig. 1. Language

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.

We refer to abstractions, booleans and pairs as data, and applications, conditionals and '$\text{let}^l\ (x, y)$ be e in e'' as consumers — thus β, δ-if and δ-let-pair reductions are data-consumptions. Flow analysis seeks a safe approximation to possible consumptions during *any* reduction of a term.

3 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 has a certain 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 [2].

The set of formulae 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

Formulae:

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

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

Subtype Relation:

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

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

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

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

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

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

Annotated Type Rules:

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

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

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

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

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

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

Fig. 2. Intersection Flow Analysis

$L \subseteq \mathcal{L}_e$ (called an *annotation*) to each type constructor. Furthermore, we allow *intersections* of properties. Note that, due to our requirement that the language is well-typed under simple (standard) typing, the individual components of an intersection will have exactly the same underlying type structure.

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- resp. co-variant way. Finally, we have three distribution rules for $\rightarrow$ and $\times$ over intersections. Note, that these three rules introduce equivalences since the opposite subtypings are derivable.

For each t, note that $\mathcal{K}(t)/=$ is finite (where $=$ is the equivalence induced by $\leq$). For every expression e and standard type t, there exists a bottom element in $\mathcal{K}(t)$ with no intersections and all positively occurring annotation equal to $\{\}$ and all negatively occurring annotation equal to $\mathcal{L}_e$.[1] Call this type $\bot_t$. For each t introduce a special constant, which we also call $\bot_t$, of this type.

Finally, figure 2 presents the inference rules defining our flow analysis. The syntax directed rules resemble the standard type rules — 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 $\kappa \wedge \kappa'$.

We remark that the following rule is admissible:

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

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

A *flow relation* Φ for an expression e is a relation between consumers and data $\Phi \subseteq \mathcal{L}_e \times \mathcal{L}_e$. 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. We define a function $\mathcal{F}$ from flow derivations $\mathcal{T}$ for e to flow relations as follows: let $\mathcal{F}(\mathcal{T}) = \Phi$ where $\Phi \subseteq \mathcal{L}_e \times \mathcal{L}_e$ is the least relation such that whenever one of the rules

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

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

is an inference in $\mathcal{T}$, then $(l, l') \in \Phi$ for all $l' \in L$.

[1] Positive and negative occurrences are defined as follows: 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 $\rightarrow$) follows the argument branch of $\rightarrow$ constructors an even (odd) number of times then ℓ is said to occur positively (negatively) in κ.

4 Decidability

We will give a syntax directed version of our inference system. Coppo, Dezani-Ciancaglini and Veneri [4] and van Bakel [32] have used a similar technique of integrating the 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:

$$\text{Sub'}\ \frac{\forall i \in I : \quad A \vdash' e : \kappa_i \quad \vdash \kappa_i \leq \kappa'_i}{A \vdash' e : \bigwedge_{i \in I} \kappa'_i}$$

Lemma 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 only rule (Sub').*

2. If Sub' $\frac{\forall i \in I : \quad A \vdash' e : \kappa_i \quad \vdash \kappa_i \leq \kappa'_i}{A \vdash' e : \bigwedge_{i \in I} \kappa'_i}$ then $A \vdash e : \bigwedge_i \kappa'_i$ can be inferred using rules (Sub) and ($\wedge$-I) from the same assumptions.

Proof. Point *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 point *2.*. □

The resulting system is sound and complete w.r.t. the original system:

Proposition 2. *Soundness: If $\frac{\mathcal{T}}{A \vdash' e : \kappa}$ is a valid derivation then there exists a valid derivation $\frac{\mathcal{T}'}{A \vdash e : \kappa}$ such that $\mathcal{F}(\frac{\mathcal{T}'}{A \vdash e : \kappa}) = \mathcal{F}(\frac{\mathcal{T}}{A \vdash' e : \kappa})$.*

Completeness: If $\frac{\mathcal{T}}{A \vdash e : \kappa}$ is a valid derivation then there exists a valid derivation $\frac{\mathcal{T}'}{A \vdash' e : \kappa}$ such that $\mathcal{F}(\frac{\mathcal{T}'}{A \vdash' e : \kappa}) \subseteq \mathcal{F}(\frac{\mathcal{T}}{A \vdash e : \kappa})$.

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

Define function $\mathit{normalize} : \mathcal{K}(t) \to \mathcal{K}(t)$ for all t as follows: $\mathit{normalize}(\kappa)$ is the result of exhaustively applying the following rewrite rules to κ:

$$\begin{array}{rl}
\kappa_1 \to^L (\kappa_2 \wedge \kappa'_2) & \longrightarrow (\kappa_1 \to^L \kappa_2) \wedge (\kappa_1 \to^L \kappa'_2) \\
(\kappa_1 \wedge \kappa'_1) \times^L \kappa_2 & \longrightarrow (\kappa_1 \times^L \kappa_2) \wedge (\kappa'_1 \times^L \kappa_2) \\
\kappa_1 \times^L (\kappa_2 \wedge \kappa'_2) & \longrightarrow (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1 \times^L \kappa'_2) \\
K[\kappa] & \longrightarrow K[\kappa'] \qquad \text{if } \kappa \longrightarrow \kappa'
\end{array}$$

where type contexts K are defined by

$$K ::= [\,] \mid \kappa \times^L K \mid K \times^L \kappa \mid \kappa \to^L K$$

. Note that $\vdash \kappa = \mathit{normalize}(\kappa)$ for all κ and that the only conjunction in "Rank 1 position" in $\mathit{normalize}(\kappa)$ is at top level.

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

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

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

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

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

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

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

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

Fig. 3. Syntax Directed Intersection Flow Analysis

Lemma 3. *If* $\frac{\mathcal{T}}{A, x : \kappa_1 \vdash^n e : \kappa}$ *is a valid derivation and* $\vdash \kappa_2 \leq \kappa_1$ *then there exists* $\mathcal{T}', \kappa'$ *such that* $\frac{\mathcal{T}'}{A, x : \kappa_2 \vdash^n e : \kappa'}$ *is a valid derivation and* $\mathcal{F}(\frac{\mathcal{T}'}{A, x : \kappa_2 \vdash^n e : \kappa'}) \subseteq \mathcal{F}(\frac{\mathcal{T}}{A, x : \kappa_1 \vdash^n e : \kappa})$.

Proof. By induction over the structure of e.

The syntax directed system is sound and complete w.r.t. the original system in the following sense:

Theorem 4. *Soundness: If* $\frac{\mathcal{T}}{A \vdash^n e : \kappa}$ *is a valid normalised derivation then there exists a valid derivation* $\frac{\mathcal{T}'}{A' \vdash e : \kappa'}$ *such that* $\mathcal{F}(\frac{\mathcal{T}'}{A' \vdash e : \kappa'}) = \mathcal{F}(\frac{\mathcal{T}}{A \vdash^n e : \kappa})$.

Completeness: If $\frac{\mathcal{T}}{A \vdash e : \kappa}$ *is a valid derivation then there exists a valid normalised derivation* $\frac{\mathcal{T}'}{A' \vdash^n e : \kappa'}$ *such that* $\mathcal{F}(\frac{\mathcal{T}'}{A' \vdash^n e : \kappa'}) \subseteq \mathcal{F}(\frac{\mathcal{T}}{A \vdash e : \kappa})$.

Proof. We prove the theorem for $\vdash'$ instead of $\vdash$. By proposition 2 the theorem follows.

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 $\dfrac{\mathcal{T}}{A \vdash' e : \kappa}$ is a valid derivation and $\mathit{normalize}(\kappa) = \bigwedge_{i \in I} \kappa_i$ then there exists a family $\dfrac{\mathcal{T}_i}{A \vdash^n e : \kappa'_i}$ for $i \in I$ of derivations such that $\vdash \kappa'_i \leq \kappa_i$ for all i.

We prove completeness by induction over the inference tree for $\dfrac{\mathcal{T}}{A \vdash' e : \kappa}$:

(Id) Trivial

($\to$-I) Assume a derivation

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

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

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

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

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

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

$$\vdash \kappa \to^{\{l\}} \kappa'_i \leq \kappa \to^{\{l\}} \kappa_i$$

for all $i \in I$

($\to$-E) Assume

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

Further, assume

$$\mathit{normalize}(\kappa' \to^L \kappa) = \bigwedge_{i \in I} (\kappa' \to^L \kappa_i)$$
$$\mathit{normalize}(\kappa') = \bigwedge_{j \in J} \kappa'_j$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash^n e : \kappa'_i \to^{L_i} \kappa''_i} \qquad \text{and} \qquad \frac{\mathcal{T}'_j}{A \vdash^n e' : \kappa'''_j}$$

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

$$\vdash \kappa_i' \to^{L_i} \kappa_i'' \leq \kappa' \to^{L} \kappa_i \qquad \text{and} \qquad \vdash \kappa_j''' \leq \kappa_j'$$

We construct

$$\frac{\mathcal{T}_i}{A \vdash^n e : \kappa_i' \to^{L_i} \kappa_i''} \quad \forall j \in J : \frac{\mathcal{T}_j'}{A \vdash^n e' : \kappa_j'''} \quad \vdash \bigwedge_{j \in J} \kappa_j''' \leq \kappa_i'$$
$$\frac{}{A \vdash^n e @^l e' : \kappa_i''}$$

where $\vdash \bigwedge_{j \in J} \kappa_j''' \leq \kappa_i'$ follows from $\vdash \kappa_j''' \leq \kappa_j'$, $\vdash \bigwedge_{j \in J} \kappa_j' = \kappa'$ and $\vdash \kappa' \leq \kappa_i'$.

We have that $\mathit{normalize}(\kappa) = \bigwedge_{i \in I} \kappa_i$ and that $\vdash \kappa_i'' \leq \kappa_i$.

($\times$-I) Assume

$$\times\text{-I}\ \frac{\dfrac{\mathcal{T}}{A \vdash' e : \kappa} \quad \dfrac{\mathcal{T}'}{A \vdash' e' : \kappa'}}{A \vdash' (e, e')^l : \kappa \times^{\{l\}} \kappa'}$$

Further, assume

$$\mathit{normalize}(\kappa) = \textstyle\bigwedge_{i \in I}(\kappa_i)$$
$$\mathit{normalize}(\kappa') = \textstyle\bigwedge_{j \in J} \kappa_j'$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash^n e : \kappa_i''} \qquad \text{and} \qquad \frac{\mathcal{T}_j'}{A \vdash^n e' : \kappa_j'''}$$

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

$$\vdash \kappa_i'' \leq \kappa_i \qquad \text{and} \qquad \vdash \kappa_j''' \leq \kappa_j'$$

Now $\mathit{normalize}(\kappa \times^{\{l\}} \kappa') = \bigwedge_{i \in I, j \in J}(\kappa_i \times^{\{l\}} \kappa_j')$ and for each $(i, j) \in I \times J$ we have

$$\frac{\dfrac{\mathcal{T}_i}{A \vdash^n e : \kappa_i''} \quad \dfrac{\mathcal{T}_j'}{A \vdash^n e' : \kappa_j'''}}{A \vdash^n (e, e')^l : \kappa_i'' \times^{\{l\}} \kappa_j'''}$$

and clearly $\vdash \kappa_i'' \times^{\{l\}} \kappa_j''' \leq \kappa_i \times^{\{l\}} \kappa_j'$.

($\times$-E) Assume

$$\times\text{-E}\ \frac{\dfrac{\mathcal{T}}{A \vdash' e : \kappa_x \times^{L} \kappa_y} \quad \dfrac{\mathcal{T}'}{A, x : \kappa_x, y : \kappa_y \vdash' e' : \kappa}}{A \vdash' \mathrm{let}^l\ (x, y)\ \mathrm{be}\ e\ \mathrm{in}\ e' : \kappa}$$

Further, assume

$$\mathit{normalize}(\kappa_x \times^{L} \kappa_y) = \textstyle\bigwedge_{i \in I}(\kappa_{ix} \times^{L} \kappa_{iy})$$
$$\mathit{normalize}(\kappa) = \textstyle\bigwedge_{j \in J} \kappa_j$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash^n e : \kappa'_{ix} \times^{L_i} \kappa'_{iy}} \qquad \text{and} \qquad \frac{\mathcal{T}'_j}{A, x : \kappa_x, y : \kappa_y \vdash^n e' : \kappa'_j}$$

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

$$\vdash \kappa'_{ix} \times^{L_i} \kappa'_{iy} \leq \kappa_{ix} \times^{L} \kappa_{iy} \qquad \text{and} \qquad \vdash \kappa'_j \leq \kappa_j$$

It follows that $\vdash \kappa'_{ix} \leq \kappa_{ix}$ and $\vdash \kappa'_{iy} \leq \kappa_{iy}$ and hence

$$\frac{\dfrac{\mathcal{T}_i}{A \vdash^n e : \kappa'_{ix} \times^{L_i} \kappa'_{iy}} \quad \dfrac{\mathcal{T}'_j}{A, x : \bigwedge_{i \in I} \kappa'_{ix}, y : \bigwedge_{i \in I} \kappa'_{iy} \vdash^n e' : \kappa'_j}}{A \vdash^n \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa'_j}$$

for $j \in J$ is the wanted family of derivations. Since $\vdash \bigwedge_{i \in I} \kappa'_{ix} \leq \kappa_x$ and $\vdash \bigwedge_{i \in I} \kappa'_{iy} \leq \kappa_y$, the use of strengthened assumptions in $\mathcal{T}'_j$ is justified by lemma 3.

(Bool-I) Trivial.

(Bool-E) Assume

$$\text{Bool-E}\ \frac{\dfrac{\mathcal{T}}{A \vdash' e : \text{Bool}^L} \quad \dfrac{\mathcal{T}}{A \vdash' e' : \kappa} \quad \dfrac{\mathcal{T}}{A \vdash' e'' : \kappa}}{A \vdash' \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

Further, assume

$$\begin{aligned} \mathit{normalize}(\text{Bool}^L) &= \text{Bool}^L \\ \mathit{normalize}(\kappa) &= \textstyle\bigwedge_{i \in I} \kappa_i \end{aligned}$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}'''}{A \vdash^n e : \text{Bool}^L} \qquad \frac{\mathcal{T}_i}{A \vdash^n e' : \kappa'_i} \qquad \text{and} \qquad \frac{\mathcal{T}'_i}{A \vdash^n e'' : \kappa''_i}$$

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

$$\vdash \kappa'_i \leq \kappa_i \qquad \text{and} \qquad \vdash \kappa''_i \leq \kappa_i$$

Now construct

$$\frac{\mathcal{T}'''}{A \vdash^n e : \text{Bool}^L} \qquad \frac{\mathcal{T}_i}{A \vdash^n e' : \kappa'_i} \qquad \frac{\mathcal{T}'_i}{A \vdash^n e'' : \kappa''_i} \qquad \vdash \kappa'_i \leq \kappa_i \qquad \vdash \kappa''_i \leq \kappa_i$$
$$\overline{\qquad\qquad\qquad\qquad A \vdash^n \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa_i}$$

(fix) Assume

$$\text{fix}\ \frac{\dfrac{\mathcal{T}}{A, x : \kappa \vdash' e : \kappa}}{A \vdash' \text{fix}^l x.e : \kappa}$$

Assume that $\mathit{normalize}(\kappa) = \bigwedge_{i \in I} \kappa_i$. Then by induction there exists a family of derivations

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

such that $\vdash \kappa_i' \leq \kappa_i$ for all i. It follows that $\vdash \bigwedge_{i \in I} \kappa_i' \leq \kappa$ and hence we have the following family of derivations:

$$\text{fix}\ \frac{\forall i \in I \dfrac{\mathcal{T}_i}{A, x : \kappa \vdash' e : \kappa_i'} \qquad \vdash \bigwedge_{i \in I} \kappa_i' \leq \kappa}{A \vdash' \text{fix}^l x.e : \kappa_j}$$

for $j \in I$.

(Sub') Assume

$$\frac{\forall i \in I : \quad \dfrac{\mathcal{T}_i}{A \vdash' e : \kappa_i} \qquad \vdash \kappa_i \leq \kappa_i'}{A \vdash' e : \bigwedge_{i \in I} \kappa_i'}$$

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

$$\frac{\mathcal{T}_{ij}}{A \vdash^n e : \kappa_{ij}'}$$

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

□

We can now argue decidability as follows: first note that the subtype relation $\vdash \kappa \leq \kappa'$ is decidable. Now given an expression e, the height of the normalised inference tree (leaving out the $\vdash \kappa \leq \kappa'$ judgements) is bounded by the size of e. W.r.t. 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 $\mathcal{K}(t)/ =$ where $t = |\kappa_i'|$.
2. Similarly, in the (fix) rule we have that the number of assumptions $A, x : \kappa \vdash_n^\wedge e' : \kappa_i$ is bounded by the size of $\mathcal{K}(t)/ =$ where $t = |\kappa_i|$.

5 Minimality

Define a *vectorising* function *vec* on properties as follows

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

where $\cap$ is pointwise set intersection and $+\!\!+$ is vector concatenation. Define an ordering $\preceq$ on properties

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

(where $\subseteq$ is pointwise subset inclusion). This is extended to judgements by defining $(x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash e : \kappa) \preceq (x_1 : \kappa'_1, \cdots, x_n : \kappa'_n \vdash e : \kappa')$ iff $\kappa \preceq \kappa'$ and $\kappa_i \preceq \kappa'_i$ for $i \in \{1, \cdots, n\}$. We say $\mathcal{T} \preceq \mathcal{T}'$ if $\mathcal{T}$ and $\mathcal{T}'$ are derivations for the same expression e, and for all subexpressions e' of e we have that for the last (closest to the conclusion) judgements $A \vdash e' : \kappa$ and $A' \vdash e' : \kappa'$ (in $\mathcal{T}$ resp. $\mathcal{T}'$) it holds that $(A \vdash e' : \kappa) \preceq (A' \vdash e' : \kappa')$.

Define $\kappa \sqcap \kappa'$ as follows

$$\begin{aligned}
\mathrm{Bool}^{L_1} \sqcap \mathrm{Bool}^{L_2} &= \mathrm{Bool}^{L_1 \cap L_2} \\
\kappa_1 \times^{L_1} \kappa'_1 \sqcap \kappa_2 \times^{L_2} \kappa'_2 &= (\kappa_1 \sqcap \kappa_2) \times^{L_1 \cap L_2} (\kappa'_1 \sqcap \kappa'_2) \\
\kappa_1 \to^{L_1} \kappa'_1 \sqcap \kappa_2 \to^{L_2} \kappa'_2 &= (\kappa_1 \sqcap \kappa_2) \to^{L_1 \cap L_2} (\kappa'_1 \sqcap \kappa'_2) \\
(\kappa_1 \wedge \kappa'_1) \sqcap \kappa_2 &= (\kappa_1 \sqcap \kappa_2) \wedge (\kappa'_1 \sqcap \kappa_2) \\
\kappa_1 \sqcap \kappa_2 &= \kappa_2 \sqcap \kappa_1
\end{aligned}$$

It is easy to check that $\sqcap$ is the greatest lower bound operator on the domain $(\mathcal{K}/=, \preceq)$. Define

$$\begin{aligned}
&(x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash e : \kappa) \sqcap (x_1 : \kappa'_1, \cdots, x_n : \kappa'_n \vdash e : \kappa') \\
&\quad = (x_1 : \kappa_1 \sqcap \kappa'_1, \cdots, x_n : \kappa_n \sqcap \kappa'_n \vdash e : \kappa \sqcap \kappa')
\end{aligned}$$

and for two derivations $\mathcal{T}$ and $\mathcal{T}'$ for the same expression e let $\mathcal{T} \sqcap \mathcal{T}'$ be a derivation such that the last judgement for any subexpression e' is the $\sqcap$ of the similar judgements in $\mathcal{T}$ and $\mathcal{T}'$. The following lemma shows that such a derivation exists:

Lemma 5. *If* $\dfrac{\mathcal{T}}{A \vdash e : \kappa}$ *and* $\dfrac{\mathcal{T}'}{A' \vdash e : \kappa'}$ *are derivations then so is* $\dfrac{\mathcal{T} \sqcap \mathcal{T}'}{A \sqcap A' \vdash e : \kappa \sqcap \kappa'}$.

Proof. First prove that $\vdash \kappa_1 \leq \kappa_2$ implies $\vdash \kappa_1 \sqcap \kappa_3 \leq \kappa_2 \sqcap \kappa_3$ for all κ_1, κ_2 and κ_3. This follows by induction on the derivation of $\vdash \kappa_1 \leq \kappa_2$. Now the lemma follows by induction over the sum of the heights of the two derivations. □

It follows as an immediate consequence that

Corollary 6. *For each* e *there exists a* minimal *derivation under the* $\preceq$ *ordering.*

If $\mathcal{T}$ is minimal for e then $\mathcal{F}(\mathcal{T})$ is the minimal flow relation derivable for e.

6 Non-Standard Semantics

In this section we give a non-standard semantics which 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, this redex will be reduced by the semantics.

Intuitively, the intersection based analysis loses information whenever computation is discarded. E.g. analysing $\text{if}^{l_1}\ \text{True}^{l_2}$ then False^{l_3} 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 computation when 'if' is reduced, we introduce a new construct 'either'. To avoid that function arguments, which are not used, are discarded, we introduce a special syntactic construct 'discard'. The type rules for the extensions of the language look as follows:

$$\text{Either}\ \frac{A \vdash e : t \quad A \vdash e' : t}{A \vdash \text{either}\ e\ \text{or}\ e' : t} \qquad \text{Discard}\ \frac{A \vdash e' : \kappa' \quad A \vdash e : \kappa}{A \vdash \text{discard}\ e'\ \text{in}\ e : \kappa}$$

The analysis is also imprecise in another way: it can predict redexes that are "blocked". E.g. for $(\text{let}^{l_1}\ (x_1, x_2)\ \text{be}\ y\ \text{in}\ \lambda^{l_2} z.z)@^{l_3}\text{False}^{l_4}$, 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 behaviour 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).

Extending the analysis to handle the new constructs is the straightforward:

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

We can show (see [20]) the following proposition on normal-forms

Proposition 7. *Any expression v such that no e exists with $v \longrightarrow_s e$ is called a* value. *An expression v is a value if and only if it has the following syntax:*

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

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 realise that e cannot be $\text{fix}\, x.e$.

True: Is clearly a value.
False: Is clearly a value.

Contexts:

$$C ::= [\,] \mid \lambda^l x.C \mid C@^l e \mid e@^l C \mid \text{if}^l\ C \text{ then } e' \text{ else } e'' \mid \text{if}^l\ e \text{ then } C \text{ else } e'' \mid \text{if}^l\ e \text{ then } e' \text{ else } C \mid (C, e')^l \mid (e, C)^l \mid \text{let}^l\ (x, y) \text{ be } C \text{ in } e' \mid \text{let}^l\ (x, y) \text{ be } e \text{ in } C \mid \text{discard}^l\ C \text{ in } e' \mid \text{discard}^l\ e \text{ in } C \mid \text{either}^l\ C \text{ or } e \mid \text{either}^l\ e \text{ or } C$$

Reduction Rules:

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

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\ e_1 \text{ or } e_2)@^{l'} e_3 \\
 & \quad \longrightarrow_s \text{either}^l\ (e_1@^{l'} e_3) \text{ or } (e_2@^{l'} e_3) \\
 & \text{let}^{l'}\ (x, y) \text{ be } (\text{either}^l\ e_2 \text{ or } e_3) \text{ in } e_1 \\
 & \quad \longrightarrow_s \text{either}^l\ (\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\ e_1 \text{ or } e_2) \text{ then } e_3 \text{ else } e_4 \\
 & \quad \longrightarrow_s \text{either}^l\ (\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. Non-Standard Reduction

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

x: Is clearly a value.

$\lambda x.e'$: 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''): 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 rule would be applicable. Since e must be well typed, the only options left are applications and variables.

□

It is not hard to show the following properties for any e:

1. For any two non-standard reduction sequences reducing e to a value v, the same set of redexes are reduced.
2. If $e \longrightarrow_s^* e'$ reduces a redex (l, l') then there exists a reduction sequence using the standard rules plus (pair) and (if) context propagation rules reducing (l, l').

7 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 judgement for the whole expression preserved, but the flow computed by the derivation is preserved. The proof is standard and can be found in [20].

We first prove an intensional version of the substitution lemma:

Lemma 8 (Substitution lemma). *If* $\dfrac{\mathcal{T}}{A, x : \kappa' \vdash e : \kappa}$ *and* $\dfrac{\mathcal{T}'}{A \vdash e' : \kappa'}$ *then there exists* $\mathcal{T}''$ *such that*

1. $\dfrac{\mathcal{T}''}{A \vdash e[e'/x] : \kappa}$ *and*

2. $\mathcal{F}(\frac{\mathcal{T}''}{A \vdash e[e'/x] : \kappa}) = \mathcal{F}(\frac{\mathcal{T}}{A, x : \kappa' \vdash e : \kappa}) \cup \mathcal{F}(\frac{\mathcal{T}'}{A \vdash e' : \kappa'})$

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

Theorem 9 (Subject Reduction). *If* $\frac{\mathcal{T}}{A \vdash e : \kappa}$ *and* $e \longrightarrow_s e'$ *then there exists* $\mathcal{T}'$ *such that*

1. $\frac{\mathcal{T}'}{A \vdash e' : \kappa}$
2. *If the reduction lets consumer* l *consume data* l' *then* $\mathcal{F}(\frac{\mathcal{T}}{A \vdash e : \kappa}) = \mathcal{F}(\frac{\mathcal{T}'}{A \vdash e' : \kappa}) \cup \{(l, l')\}$ *and*
3. *If the reduction is a non-consumption then* $\mathcal{F}(\frac{\mathcal{T}'}{A \vdash e' : \kappa}) = \mathcal{F}(\frac{\mathcal{T}}{A \vdash e : \kappa})$.

Proof. The interesting cases follow from lemma 8. □

8 Completeness

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*, i.e. the computed flow information is preserved.

Lemma 10. *Let* e *be an expression with* $n > 0$ *occurrences of a variable* x*. If* $\frac{\mathcal{T}}{A \vdash e[e'/x] : \kappa}$ *then there exists* $\kappa_1 \cdots \kappa_n, \mathcal{T}'$ *and* $\mathcal{T}_1 \cdots \mathcal{T}_n$ *s.t.*

1. $\frac{\mathcal{T}_i}{A \vdash e' : \kappa_i}$
2. $\frac{\mathcal{T}'}{A, x : \bigwedge_{i \in \{1,\cdots,n\}} \kappa_i \vdash e : \kappa}$ *and*
3. $\mathcal{F}(\frac{\mathcal{T}}{A \vdash e[e'/x] : \kappa}) = \bigcup_i \mathcal{F}(\frac{\mathcal{T}_i}{A \vdash e' : \kappa_i}) \cup \mathcal{F}(\frac{\mathcal{T}'}{A, x : \bigwedge_{i \in \{1,\cdots,n\}} \kappa_i \vdash e : \kappa})$

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

Theorem 11 (Subject Expansion). *If* $\frac{\mathcal{T}'}{A \vdash e' : \kappa}$ *and* $e \longrightarrow_s e'$ *then there exists* $\mathcal{T}$ *such that*

1. $\frac{\mathcal{T}}{A \vdash e : \kappa}$
2. *If the reduction lets consumer* l *consume data* l' *then* $\mathcal{F}(\frac{\mathcal{T}}{A \vdash e : \kappa}) = \mathcal{F}(\frac{\mathcal{T}'}{A \vdash e' : \kappa}) \cup \{(l, l')\}$ *and*
3. *If the reduction is a non-consumption then* $\mathcal{F}(\frac{\mathcal{T}'}{A \vdash e' : \kappa}) = \mathcal{F}(\frac{\mathcal{T}}{A \vdash e : \kappa})$.

If $\mathcal{T}$ is a derivation where x does not occur, we write $\mathcal{T}, x : \kappa$ for the derivation where $x : \kappa$ is added to all environments: if $\mathcal{T}$ is a valid derivation and x does not occur free in the derivation then $\mathcal{T}, x : \kappa$ is a valid derivation.

Proof. Induction over the definition of $\longrightarrow_s$, i.e. the β and δ rules are the base cases and the context rule is the induction step:

(β) Two cases:

1. Assume x not free in e. Then $(\lambda^{l'} x.e)@^l e' \longrightarrow_s \text{discard}^l\ e'$ in e. We have the following inference tree:

$$\frac{\dfrac{\mathcal{T}'}{A \vdash e' : \kappa'} \qquad \dfrac{\mathcal{T}}{A \vdash e : \kappa}}{A \vdash \text{discard}^l\ e' \text{ in } e : \kappa}$$

   so we can clearly construct:

$$\frac{\dfrac{\dfrac{\dfrac{\mathcal{T}, x : \kappa'}{A, x : \kappa' \vdash e : \kappa}}{A \vdash \lambda^{l'} x.e : \kappa' \to^{\{l'\}} \kappa} \qquad \dfrac{\mathcal{T}'}{A \vdash e' : \kappa'}}{A \vdash (\lambda^{l'} x.e)@^l e' : \kappa}$$

   Point *2.* follows immediately.

2. Assume x has $n \geq 1$ occurrences in e. Then $(\lambda^{l'} x.e)@^l e' \longrightarrow_s e[e'/x]$. By assumption:

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

   By lemma 10 we have $\mathcal{T}_i, \kappa_i$ s.t.

$$\frac{\mathcal{T}_i}{A \vdash e' : \kappa_i} \qquad \text{and} \qquad \frac{\mathcal{T}'}{A, x : \bigwedge_i \kappa_i \vdash e : \kappa}$$

   Finally:

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

   Point *2.* follows from lemma 10.

(δ-if) Assume

$$\text{if}^l\ \text{True}^{l'}\ \text{then } e \text{ else } e' \longrightarrow_s \text{either}^l\ e \text{ or } e'$$

By assumption:

$$\frac{\dfrac{\mathcal{T}}{A \vdash e : \kappa} \qquad \dfrac{\mathcal{T}'}{A \vdash e' : \kappa}}{A \vdash \text{either}^{l}\ e \text{ or } e' : \kappa}$$

So clearly:

$$\frac{\dfrac{}{A \vdash \text{True}^{l'} : \text{Bool}^{\{l'\}}} \qquad \dfrac{\mathcal{T}}{A \vdash e : \kappa} \qquad \dfrac{\mathcal{T}'}{A \vdash e' : \kappa}}{A \vdash \text{if}^{l}\ \text{True}^{l'}\ \text{then } e \text{ else } e' : \kappa}$$

The case for False is similar.

(δ-let-pair) The four cases are handled as in the (β) case.

"Context Rule": Trivial induction on the context C.

"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) for 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:

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

where

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

$$\mathcal{T}_3 = \frac{\dfrac{\dfrac{\mathcal{T}_3'}{A \vdash e_3 : \kappa_4' \rightarrow^{L_3} \kappa_3} \qquad \dfrac{\mathcal{T}_3''}{A \vdash e_4 : \kappa_4'}}{A \vdash (e_3@^{l'} e_4) : \kappa_3} \qquad \vdash \kappa_3 \leq \kappa}{A \vdash (e_3@^{l'} e_4) : \kappa}$$

We can now construct:

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

where

$$\mathcal{T}_4 = \dfrac{\dfrac{\mathcal{T}_2'}{A \vdash e_2 : \kappa_4 \to^{L_2} \kappa_2} \qquad \dfrac{\overline{\vdash \kappa_4 \wedge \kappa_4' \leq \kappa_4} \qquad \vdash \kappa_2 \leq \kappa \quad L_2 \subseteq L_2 \cup L_3}{\vdash \kappa_4 \to^{L_2} \kappa_2 \leq (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}}{A \vdash e_2 : (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}$$

and

$$\mathcal{T}_5 = \dfrac{\dfrac{\mathcal{T}_3'}{\vdash e_3 : \kappa_4' \to^{L_3} \kappa_3} \qquad \dfrac{\overline{\vdash \kappa_4 \wedge \kappa_4' \leq \kappa_4'} \qquad \vdash \kappa_3 \leq \kappa \quad L_3 \subseteq L_2 \cup L_3}{\vdash \kappa_4' \to^{L_3} \kappa_3 \leq (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}}{\vdash e_3 : (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}$$

□

9 Handling 'fix'

The idea of this section is to show that there exists a finite unfolding e^m of any expression $\mathrm{fix}\, x.e$ 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 12).
2. Show subject expansion for $C[\mathrm{fix}\, x.e^{(m)}] \longrightarrow C[e^n]$ where $e^{(m)}$ is m times unfolding of e and e^n is n unfoldings of e and $\bot$ inserted for x (corollary 16).
3. Show subject expansion for $C[\mathrm{fix}\, x.e] \longrightarrow C[\mathrm{fix}\, x.e^{(m)}]$ (lemma 17).

Lemma 12. *Let e be an expression with $n > 0$ occurrences of x and no occurrences of y. Let*

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

be minimal derivations. Then

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

Proof. Trivial consequence of theorem 11. □

Definition 13. For any expression e of type t, define

$$\begin{aligned} e^0 &= \bot_t \\ e^{n+1} &= e[e^n/x] \end{aligned}$$

Lemma 14. *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 $\dfrac{\mathcal{T}}{A \vdash C[e^m] : \kappa}$ is a minimal derivation then it contains the judgements $A' \vdash e^n : \kappa'$ and $A'' \vdash e^m : \kappa''$ with $A' \mid_{FV(e)} = A'' \mid_{FV(e)}$ and $\kappa' = \kappa''$.*

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

Definition 15. For any expression e, define

$$\begin{aligned} e^{(0)} &= x \\ e^{(n+1)} &= e[e^{(n)}/x] \end{aligned}$$

Corollary 16. *Let m, n be as computed by lemma 14 and*

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

be the minimal derivation. There exists

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

such that

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

Proof. Immediate from lemma 14 and lemma 10. □

Lemma 17. *Let $\mathcal{T}$, A, κ, C and e (with exactly one occurrence of x) be given such that*

$$\frac{\mathcal{T}}{A \vdash C[\mathrm{fix}\, x.e^{(m)}] : \kappa}$$

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

$$\frac{\mathcal{T}'}{A \vdash C[\mathrm{fix}\, x.e] : \kappa} \qquad \textit{and} \qquad \mathcal{F}(\frac{\mathcal{T}'}{A \vdash C[\mathrm{fix}\, x.e] : \kappa}) \subseteq \mathcal{F}(\frac{\mathcal{T}}{A \vdash C[\mathrm{fix}\, x.e^{(m)}] : \kappa})$$

Proof. We have the following derivation:

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

Dismantle this derivation according to lemma 10 into

$$\frac{\mathcal{T}_0}{A, x : \kappa_m \vdash e : \kappa_0} \qquad \frac{\mathcal{T}_1}{A, x : \kappa_0 \vdash e : \kappa_1} \quad \cdots \quad \frac{\mathcal{T}_m}{A, x : \kappa_{m-1} \vdash e : \kappa_m}$$

By the property of strengthened assumptions, we find

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

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

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

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

We summarize corollary 16 and lemma 17 as follows (where we use lemma 12 to generalise to an arbitrary number of occurrences of the fix-bound variable).

Theorem 18. *Let $C[\mathrm{fix}^l x.e]$ be given. Then there exists m such that if*

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

then there exists $\mathcal{T}'$ such that

$$\frac{\mathcal{T}'}{A \vdash C[\mathrm{fix}^l x.e] : \kappa} \qquad \textit{and} \qquad \mathcal{F}(\frac{\mathcal{T}'}{A \vdash C[\mathrm{fix}^l x.e] : \kappa}) \subseteq \mathcal{F}(\frac{\mathcal{T}}{A \vdash C[e^m] : \kappa})$$

10 Normal Forms

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

Theorem 19. *If v is a value, there exists $A, \kappa, \mathcal{T}$ such that $\frac{\mathcal{T}}{A \vdash v : \kappa}$ and $\mathcal{F}(\frac{\mathcal{T}}{A \vdash v : \kappa}) = \emptyset$.*

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 $A(x)$ resp. $A'(x)$ if x is not in the domain of A' resp. not in the domain of A. We extend this to use the abbreviation $\mathcal{T} \wedge A$ for the derivation where A is intersected with all environments in $\mathcal{T}$ (this derivation contains appropriate subsumption steps at variable occurrences). Clearly $\mathcal{T} \wedge A$ is a valid derivation if $\mathcal{T}$ is and $\mathcal{F}(\mathcal{T}) = \mathcal{F}(\mathcal{T} \wedge A)$.

Proof. Call a property $\kappa \in \mathcal{K}$ *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 $\{\}$.

We will show that that

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

It follows from point 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) is the empty set.

The proof proceeds by induction over the structure of values.

True^l: Any A, κ will do.
False^l: Any A, κ will do.
x: Let κ be the given result empty type. Then any $(A, x : \kappa)$ will do.
$\lambda^l x.v$: By induction we have

$$\frac{\mathcal{T}}{A, x : \kappa \vdash v : \kappa'}$$

where A and κ are result empty. If v is not a variable or an application, κ' is argument empty. Otherwise, the above is true for *any* result-empty κ' in particular the one that is also argument empty. Thus $\kappa \to^L \kappa'$ is argument-empty for any L. So

$$\frac{\dfrac{\mathcal{T}}{A, x : \kappa \vdash v : \kappa'}}{A \vdash \lambda^l x.v : \kappa \to^L \kappa'}$$

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

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

where A' is result-empty and κ' 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. Then by induction there is

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

where A is result-empty.
We now construct

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

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 $A, \mathcal{T}$ s.t.

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

and $A', A'', \kappa', \kappa'', \mathcal{T}', \mathcal{T}''$ s.t.

$$\frac{\mathcal{T}'}{A' \vdash v' : \kappa'} \qquad \text{and} \qquad \frac{\mathcal{T}''}{A'' \vdash v'' : \kappa''}$$

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

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

either^l v or v': Similar, but simpler than the 'if' case.
$(v, v')^l$: Follows by simple induction.
let^l (x, y) be $\overline{v}$ in v': We have by induction

$$\frac{\mathcal{T}'}{A', x : \kappa_x, y : \kappa_y \vdash v' : \kappa'}$$

where $A', x : \kappa_x, y : \kappa_y$ is result-empty and κ argument-empty (if v is an application or a variable we choose the argument- and result-empty κ as above).
For the result-empty $\kappa_x \times^{\{\}} \kappa_y$ we have

$$\frac{\mathcal{T}}{A \vdash \overline{v} : \kappa_x \times^{\{\}} \kappa_y}$$

where A is result-empty.
We conclude

$$\frac{\dfrac{\mathcal{T} \wedge A'}{A \wedge A' \vdash \overline{v} : \kappa_x \to^{\{\}} \kappa_y} \qquad \dfrac{\mathcal{T}' \wedge A}{(A \wedge A'), x : \kappa_x, y : \kappa_y \vdash v' : \kappa'}}{A \wedge A' \vdash \text{let}^l\ (x, y) \text{ be } \overline{v} \text{ in } v' : \kappa'}$$

where clearly $A \wedge A'$ is result-empty and κ result empty.

discard^l v in v': Simple induction.
$\bot_t$: Clearly, $A \vdash \bot_t : \kappa$ for any A, κ.

□

11 Summarising the Results

Sections 8, 9 and 10 prove that the analysis is exact under non-standard reduction: let e be any expression, apply theorem 18 exhaustively yielding a fix-free term e'. By theorem 9 and theorem 18, the minimal predictable flow of e and e' is identical. By inductively applying theorem 11, 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 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. (Note that the context propagation rules are necessary since e.g. (let (a, b) be x in $\lambda y.y$)@True is a value under standard reduction — for closed terms, context propagation rules are not necessary). We can summarize:

Theorem 20 (Exactness). *Let e be any expression and let $\mathcal{T}$ be the minimal derivation for e. Then for any redex (l, l') in $\mathcal{F}(\mathcal{T})$ there exists a reduction sequence using standard reduction plus the context propagation rules (pair) and (if) such that (l, l') is reduced.*

12 Related Work

12.1 Intersection Types

It is well-known that the intersection type discipline gives an exact characterisation of normalising lambda-terms [4,2] — 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.

12.2 Flow Analysis

Flow analysis as discussed in this paper is described by Sestoft [26] (who coined his analysis *closure analysis*) and Shivers [27] (who coined his family of analyses *kCFA*). Palsberg gave a constraint formulation of closure analysis [24]. It is shown in [21] that, in contrast to popular belief, 0CFA differs from closure analysis: Shivers' family of analyses is *evaluation-order* dependent[2].

Palsberg and O'Keefe showed that the type information derived using closure analysis corresponds to Amadio-Cardelli typing [25]. Independently, Heintze showed the same result and the converse [8]: by annotating Amadio-Cardelli typings, closure analysis is derived.

[2] 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.

Heintze and McAllester define a variant of Palsberg's constraint formulation of closure analysis [9]. Termination of the analysis relies on the fact that the analysed program is well-typed — if the the size of types is bounded the analysis will run in quadratic time (in contrast to Palsberg's cubic algorithm).

Shivers' analyses are abstractions of an instrumented semantics. This semantics is defined using *contours* that keep track of different live instances of variables — kCFA 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 [13] and by Nielson and Nielson [22]. Both allows 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 [1]. The goal of his work is practical (modularity and increased precision compared to closure analysis), but an assessment of the practicality (in particular complexity) of his analysis is made difficult by the lack of a clearcut 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 describes a family of flow analyses based on types [21]: 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 [6].

12.3 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 an instance of annotated types due to the intersection). Examples of Church style analyses are: binding-time analysis [23,12,5], strictness analysis [16,17,33,14,3], boxing analysis [18,11,15], totality analysis [28,29] and flow analysis [8,21,6]. Effect systems also belong to this category [19,31].

In *Curry style* analysis, the analysis does not make 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 [7], dynamic typing [10] and flow analysis [1,9]

13 Conclusion

We have presented a flow analysis for a higher-order typed language with recursion. The analysis is proven to be 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 (by a theorem by Statman [30]) 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

1. A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *International Conference on Functional Programming*. ACM Press, 1997. To Appear.
2. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 1983.
3. N. Benton. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, University of Cambridge, December 1992.
4. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.
5. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Proc. 2nd Int'l Static Analysis Symposium (SAS)*, volume 983 of *LNCS*, pages 118–135, 1995.
6. K.-F. Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Royal Institute of Technology, Sweden, 1997.
7. C. Gomard. Higher order partial evaluation - hope for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, September 1989.
8. N. Heintze. Control-flow analysis and type systems. In *Symposium on Static Analysis (SAS)*, volume 983 of *LNCS*, pages 189–206, Glasgow, 1995.
9. N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. In *Programming Language Design and Implementation (PLDI)*, 1997.
10. F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)*, 22(3):197–230, 1994.
11. F. Henglein and J. Jørgensen. Formally optimal boxing. In *Proc. 21st ACM Symp. on Principles of Programming Languages (POPL), Portland, Oregon*. ACM, 1994.
12. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, Apr. 1994.
13. S. Jaganathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL'95*, pages 393–407. ACM Press, 1995.
14. T. Jensen. *Abstract Interpretation in Logical Form*. PhD thesis, Imperial College, Univ. of London, November 1992. Available as DIKU Report 93/11.

15. J. Jørgensen. *A Calculus for Boxing Analysis of Polymorphically Typed Languages*. PhD thesis, DIKU, University of Copenhagen, October 1995.
16. T. Kuo and P. Mishra. On strictness and its analysis. In *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pages 144–155, Jan. 1987.
17. T. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 260–272. ACM Press, Sept. 1989.
18. X. Leroy. Unboxed objects and polymorphic typing. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programmin gLanguages (POPL), Albuquerque, New Mexico*, pages 177–188. ACM Press, Jan. 1992.
19. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*. ACM, 1988.
20. C. Mossin. Exact flow analysis. Technical report, DIKU, 1997.
21. C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, January 1997.
22. F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *24th Symposium on Principles of Programming Languages*, pages 332–345, 1997.
23. H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda-calculus. In *Fifteenth ACM Symposium on Principles of Programming Languages*, pages 98–106. ACM Press, January 1988. Extended Abstract.
24. J. Palsberg. Global program analysis in constraint form. In *19th International Colloquium on Trees in Algebra and Programming (CAAP'94)*, volume 787 of *LNCS*, pages 276–290, 1994.
25. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Principles of Programming Languages*, 1995.
26. P. Sestoft. *Analysis and Efficient Implementation of Functional Languages*. PhD thesis, DIKU, University of Copenhagen, Oct. 1991.
27. O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.
28. K. L. Solberg. Strictness and totality analysis. In *Symposium on Static Analysis*, pages 408–422, 1994.
29. K. L. Solberg. Strictness and totality analysis with conjunction. In *TAPSOFT*, pages 501–515, 1995.
30. R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, Juli 1979.
31. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.
32. S. van Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151(2):385–435, 1995.
33. D. Wright. A new technique for strictness analysis. In *Proc. Int'l J. Conf. on Theory and Practice of Software Development (TAPSOFT)*, April 1991.