Faithful Translations between Polyvariant Flows and Polymorphic Types

Torben Amtoft[1] and Franklyn Turbak[2] [⋆]

[1] tamtoft@bu.edu, Boston University, Boston MA 02215, USA
[2] fturbak@wellesley.edu, Wellesley College, Wellesley MA 02481, USA

Abstract. Recent work has shown equivalences between various type systems and flow logics. Ideally, the translations upon which such equivalences are based should be *faithful* in the sense that information is not lost in round-trip translations from flows to types and back or from types to flows and back. Building on the work of Nielson & Nielson and of Palsberg & Pavlopoulou, we present the first faithful translations between a class of finitary polyvariant flow analyses and a type system supporting polymorphism in the form of intersection and union types. Additionally, our flow/type correspondence solves several open problems posed by Palsberg & Pavlopoulou: (1) it expresses call-string based polyvariance (such as k-CFA) as well as argument based polyvariance; (2) it enjoys a subject reduction property for flows as well as for types; and (3) it supports a flow-oriented perspective rather than a type-oriented one.

1 Introduction

Type systems and flow logic are two popular frameworks for specifying program analyses. While these frameworks seem rather different on the surface, both describe the "plumbing" of a program, and recent work has uncovered deep connections between them. For example, Palsberg and O'Keefe [PO95] demonstrated an equivalence between determining flow safety in the monovariant 0-CFA flow analysis and typability in a system with recursive types and subtyping [AC93]. Heintze showed equivalences between four restrictions of 0-CFA and four type systems parameterized by (1) subtyping and (2) recursive types [Hei95].

Because they merge flow information for all calls to a function, monovariant analyses are imprecise. Greater precision can be obtained via polyvariant analyses, in which functions can be analyzed in multiple abstract contexts. Examples of polyvariant analyses include call-string based approaches, such as k-CFA [Shi91,JW95,NN97], polymorphic splitting [WJ98], type-directed flow analysis [JWW97], and argument based polyvariance, such as Schmidt's analysis [Sch95] and Agesen's cartesian product analysis [Age95]. In terms of the flow/type correspondence, several forms of flow polyvariance appear to correspond to type polymorphism expressed with intersection and union types

[⋆] Both authors were supported by NSF grant EIA–9806747. This work was conducted as part of the Church Project (http://www.cs.bu.edu/groups/church/).

[Ban97,WDMT97,DMTW97,PP99]. Intuitively, intersection types are finitary polymorphic types that model the multiple analyses for a given abstract closure, while union types are finitary existential types that model the merging of abstract values where flow paths join. Palsberg and Pavlopoulou (henceforth P&P) were the first to formalize this correspondence by demonstrating an equivalence between a class of flow analyses supporting argument based polyvariance and a type system with union and intersection types [PP99].

If type and flow systems encode similar information, translations between the two should be *faithful*, in the sense that round-trip translations from flow analyses to type derivations and back (or from type derivations to flow analyses and back) should not lose precision. Faithfulness formalizes the intuitive notion that a flow analysis and its corresponding type derivation contain the same information content. Interestingly, neither the translations of Palsberg and O'Keefe nor those of P&P are faithful. The lack of faithfulness in P&P is demonstrated by a simple example. Let $e = (\lambda^1\texttt{x.succ x}) @ ((\lambda^2\texttt{y.y}) @ 3)$, where we have labeled two program points of interest. Consider an initial monovariant flow analysis in which the only abstract closure reaching point 1 is $v_1 = (\lambda\texttt{x.succ x}, [\,])$ and the only one reaching point 2 is $v_2 = (\lambda\texttt{y.y}, [\,])$. The flow-to-type translation of P&P yields the expected type derivation:

$$\dfrac{\dfrac{\cdots}{[\,] \vdash \lambda^1\texttt{x.succ x} : \texttt{int} \to \texttt{int}} \qquad \dfrac{\dfrac{\cdots}{[\,] \vdash \lambda^2\texttt{y.y} : \texttt{int} \to \texttt{int}} \quad \cdots}{[\,] \vdash (\lambda^2\texttt{y.y}) @ 3 : \texttt{int}}}{[\,] \vdash (\lambda^1\texttt{x.succ x}) @ ((\lambda^2\texttt{y.y}) @ 3) : \texttt{int}}$$

However, P&P's type-to-flow translation loses precision by merging into a single set all abstract closures associated with the same type in a given derivation. For the example derivation above, the type $\texttt{int} \to \texttt{int}$ translates back to the abstract closure set $V = \{v_1, v_2\}$, yielding a less precise flow analysis in which V flows to both points 1 and 2. In contrast, Heintze's translations are faithful. The undesirable merging in the above example is avoided by annotating function types with a label set indicating the source point of the function value. Thus, $\lambda^1\texttt{x.succ x}$ has type $\texttt{int} \overset{\{1\}}{\to} \texttt{int}$ while $\lambda^2\texttt{y.y}$ has type $\texttt{int} \overset{\{2\}}{\to} \texttt{int}$.

In this paper, we present the first faithful translations between a broad class of polyvariant flow analyses and a type system with polymorphism in the form of intersection and union types. The translations are faithful in the sense that a round-trip translation acts as the identity for canonical types/flows, and otherwise canonicalizes. In particular, our round-trip translation for types preserves non-recursive types that P&P may transform to recursive types. We achieve this result by adapting the translations of P&P to use a modified version of the flow analysis framework of Nielson and Nielson (henceforth N&N) [NN97]. As in Heintze's translations, annotations play a key role in the faithfulness of our translations: we (1) annotate flow values to indicate the sinks to which they flow, and (2) annotate union and intersection types with component labels. These annotations can be justified independently of the flow/type correspondence.

Additionally, our framework solves several open problems posed by P&P:

1. *Unifying P&P and N&N*: Whereas P&P's flow specification can readily handle only argument based polyvariance, N&N's flow specification can also express call-string based polyvariance. So our translations give the first type system corresponding to k-CFA analysis where $k \geq 1$.
2. *Subject reduction for flows*: We inherit from N&N's flow logic the property that flow information valid before a reduction step is still valid afterwards. In contrast, P&P's flow system does not have this property.
3. *Letting "flows have their way"*: P&P discuss mismatches between flow and type systems that imply the need to choose one perspective over the other when designing a translation between the two systems. P&P always let types "have their way"; for example they require analyses to be finitary and to analyze all closure bodies, even though they may be dead code. In contrast, our design also lets flows "have their way", in that our type system does not require all subexpressions to be analyzed.

Due to space limitations, the following presentation is necessarily somewhat dense. Please see the companion technical report [AT00] for a more detailed exposition with additional explanatory text, more examples, and proofs.

2 The Language

We consider a language whose core is λ-calculus with recursion:

$$
\begin{array}{l}
ue \in \mathbf{UnLabExpr} ::= z \mid \mu f.\lambda x.e \mid e \,@\, e \mid c \mid \texttt{succ}\ e \mid \texttt{if0}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \ldots \\
e \in \mathbf{LabExpr} ::= ue^{l} \quad l \in \mathbf{Lab} \quad z \in \mathbf{Var} ::= x \mid f \quad x \in \mathbf{NVar} \quad f \in \mathbf{RVar}
\end{array}
$$

$\mu f.\lambda x.e$ denotes a function with parameter x which may call itself via f; $\lambda x.e$ is a shorthand for $\mu f.\lambda x.e$ where f does not occur in e. Recursive variables (ranged over by f) and non-recursive variables (ranged over by x) are distinct; z ranges over both. There are also integer constants c, the successor function, and the ability to test for zero. Other constructs might be added, e.g., `let`[1].

All subexpressions have integer labels. We often write labels on constructors (e.g., write $\lambda^{l}x.e$ for $(\lambda x.e)^{l}$ and $e_1 \,@_l\, e_2$ for $(e_1 \,@\, e_2)^{l}$).

Example 1. The expression $\mathsf{P}_1 \equiv (\lambda^6 \mathtt{g}.((\mathtt{g}^3 \,@_2\, \mathtt{g}^4) \,@_1\, 0^5)) \,@_0\, (\lambda^8 \mathtt{x}.\mathtt{x}^7)$ shows the need for polyvariance: $\lambda^8 \mathtt{x}.\mathtt{x}^7$ is applied both to itself and to an integer.

Like N&N, but unlike P&P, we use an environment-based small step semantics. This requires incorporating N&N's `bind` and `close` constructs into our expression syntax. An expression not containing `bind` or `close` is said to be *pure*. Every abstraction body must be pure. A program P is a pure, closed expression where each label occurs at most once within P; thus each subexpression of P ($\in$ $\mathit{SubExpr}_P$) denotes a unique "position" within P.

[1] Let-polymorphism can be simulated by intersection types.

3 The Type System

Types are built from base types, function types, intersection types, and union types as follows (where **ITag** and **UTag** are unspecified):

$$\begin{array}{l}
t \in \textbf{ElementaryType} ::= \texttt{int} \mid \bigwedge_{i \in I}\{K_i : u_i \to u'_i\} \\
u \in \textbf{UnionType} ::= \bigvee_{i \in I}\{q_i : t_i\} \\
K \in \mathcal{P}(\textbf{ITag}) \qquad k \in \textbf{ITag} \qquad q \in \textbf{UTag}
\end{array}$$

Such grammars are usually interpreted inductively, but this one is to be viewed co-inductively. That is, types are regular (possibly infinite) trees formed according to the above specification. Two types are considered equal if their infinite unwindings are equal (modulo renaming of the index sets I).

An elementary type t is either an integer `int` or an intersection type of the form $\bigwedge_{i \in I}\{K_i : u_i \to u'_i\}$, where I is a (possibly empty) finite index set, each u_i and u'_i is a union type, and the K_i's, known as *I-tagsets*, are non-empty disjoint sets of *I-tags*. We write $dom(t)$ for $\cup_{i \in I} K_i$. Intuitively, if an expression e has the above intersection type then *for all* $i \in I$ it holds that the expression maps values of type u_i into values of type u'_i.

A union type u has the form $\bigvee_{i \in I}\{q_i : t_i\}$, where I is a (possibly empty) finite index set, each t_i is an elementary type, and the q_i are distinct *U-tags*. We write $dom(u)$ for $\cup_{i \in I}\{q_i\}$, and $u.q = t$ if there exists $i \in I$ such that $q = q_i$ and $t = t_i$. We assume that for all $i \in I$ it holds that $t_i = \texttt{int}$ iff $q_i = q_{\text{int}}$ where q_{int} is a distinguished U-tag. Intuitively, if an expression e has the above union type then *there exists* an $i \in I$ such that e has the elementary type t_i.

If $I = \{1 \cdots n\}$ $(n \geq 0)$, we write $\bigvee(q_1 : t_1, \cdots, q_n : t_n)$ for $\bigvee_{i \in I}\{t_i : q_i\}$ and write $\bigwedge(K_1 : u_1 \to u'_1, \cdots, K_n : u_n \to u'_n)$ for $\bigwedge_{i \in I}\{K_i : u_i \to u'_i\}$. We write u_{int} for $\bigvee(q_{\text{int}} : \texttt{int})$.

The type system is much as in P&P except for the presence of tags. These annotations serve as witnesses for existentials in the subtyping relation and play crucial roles in the faithfulness of our flow/type correspondence. U-tags track the "source" of each intersection type and help to avoid the precision-losing merging seen in P&P's type-to-flow translation (cf. Sect. 1). I-tagsets track the "sinks" of each arrow type and help to avoid unnecessary recursive types in the flow-to-type translation.

3.1 Subtyping

We define an ordering $\leq$ on union types and an ordering $\leq_\wedge$ on elementary types, where $u \leq u'$ means that u' is less precise than u and similarly for $\leq_\wedge$. To capture the intuition that something of type t_1 has one of the types t_1 or t_2, $\leq$ should satisfy $\bigvee(q_1 : t_1) \leq \bigvee(q_1 : t_1, q_2 : t_2)$. For $\leq_\wedge$, we want to capture the following intuition: a function that can be assigned both types $u_1 \to u'_1$ and $u_2 \to u'_2$ also (1) can be assigned one of them[2] and (2) can be assigned a function type

[2] I.e., for $i \in \{1, 2\}$, $\bigwedge(K_1 : u_1 \to u'_1, K_2 : u_2 \to u'_2) \leq_\wedge \bigwedge(K_i : u_i \to u'_i)$.

that "covers" both[3]. The following mutually recursive specification of $\leq$ and $\leq_\wedge$ formalizes these considerations:

$\bigvee_{i\in I}\{q_i : t_i\} \leq \bigvee_{j\in J}\{q'_j : t'_j\}$
iff for all $i \in I$ there exists $j \in J$ such that $q_i = q'_j$ and $t_i \leq_\wedge t'_j$

$\texttt{int} \leq_\wedge \texttt{int}$

$\bigwedge_{i\in I}\{K_i : u_i \to u'_i\} \leq_\wedge \bigwedge_{j\in J}\{K'_j : u''_j \to u'''_j\}$
iff for all $j \in J$ there exists $I_0 \subseteq I$ such that
$K'_j = \cup_{i\in I_0} K_i$ and $\forall i \in I_0.\ u'_i \leq u'''_j$ and
$\forall q \in \mathit{dom}(u''_j).\ \exists i \in I_0.\ q \in \mathit{dom}(u_i)$ and $u''_j.q \leq_\wedge u_i.q$.

Observe that if $t \leq_\wedge t'$, then $\mathit{dom}(t') \subseteq \mathit{dom}(t)$. The above specification is not yet a *definition* of $\leq$ and $\leq_\wedge$, since types may be infinite. However, it gives rise to a monotone functional on a complete lattice whose elements are pairs of relations; $\leq$ and $\leq_\wedge$ are then defined as the (components of) the *greatest* fixed point of this functional. Coinduction yields:

Lemma 1. *The relations $\leq$ and $\leq_\wedge$ are reflexive and transitive.*

Our subtyping relation differs from P&P's in several ways. The U-tags and I-tags serve as "witnesses" for the existential quantifiers present in the specification, reducing the need for search during type checking. Finally, our $\leq$ seems more natural that the P&P's $\leq_1$, which is not a congruence and in fact has the rather odd property that if $\vee(T_1, T_2) \leq_1 \vee(T_3, T_4)$ (with the T_i's all distinct), then either $\vee(T_1, T_2) \leq_1 T_3$ or $\vee(T_1, T_2) \leq_1 T_4$.

3.2 Typing Rules

A *typing* T for a program P is a tuple (P, IT_T, UT_T, D_T), where IT_T is a finite set of I-tags, UT_T is a finite set of U-tags, and D_T is a derivation of $[\,] \vdash P : u$ according to the inference rules given in Fig. 1. In a judgement $A \vdash e : u$, A is an environment with bindings of the form $[z \mapsto u]$; we require that all I-tags in D_T belong to IT_T and that all U-tags in D_T belong to UT_T.

Subtyping has been inlined in all of the rules to simplify the type/flow correspondence. The rules for function abstraction and function application are both instrumented with a "witness" that enables reconstructing the justification for applying the rule. In $[\mathsf{app}]^{w^@}$, the type of the operator is a (possibly empty) union, all components of which have the expected function type but the I-tagsets may differ; the app-*witness* $w^@$ is a partial mapping from $\mathit{dom}(u_1)$ that given q produces the corresponding I-tagset. In $[\mathsf{fun}]^{w^\lambda}$, the function types resulting from analyzing the body in several different environments are combined into an intersection type t. This is wrapped into a union type with an arbitrary U-tag q,

[3] I.e., $\bigwedge(K_1 : u_1 \to u'_1, K_2 : u_2 \to u'_2) \leq_\wedge \bigwedge(K_1 \cup K_2 : u_{12} \to u'_{12})$, where any value having one of the types u'_1 or u'_2 also has type u'_{12}, and where any value having type u_{12} also has one of the types u_1 or u_2.

$$\begin{array}{lll}
[\mathsf{var}] & A \vdash z^l : u & \text{if } A(z) \leq u \\[2ex]
[\mathsf{fun}]^{(q:t)} & \dfrac{\forall k \in K : A[f \mapsto u''_k, x \mapsto u_k] \vdash e : u'_k}{A \vdash \mu f.\lambda^l x.e : u} & \begin{array}{l}\text{if } t = \bigwedge_{k \in K}\{\{k\} : u_k \to u'_k\} \\ \wedge \bigvee(q : t) \leq u \\ \wedge\, \forall k \in K.\ \bigvee(q : t) \leq u''_k\end{array} \\[4ex]
[\mathsf{app}]^{w^@} & \dfrac{A \vdash e_1 : u_1 \quad A \vdash e_2 : u_2}{A \vdash e_1 @_l e_2 : u} & \text{if } \forall q \in \mathit{dom}(u_1).\, u_1.q \leq_\wedge \bigwedge(w^@(q) : u_2 \to u) \\[2ex]
[\mathsf{con}] & A \vdash c^l : u & \text{if } u_{\text{int}} \leq u \\[2ex]
[\mathsf{suc}] & \dfrac{A \vdash e_1 : u_1}{A \vdash \mathtt{succ}^l\, e_1 : u} & \text{if } u_1 \leq u_{\text{int}} \leq u \\[2ex]
[\mathsf{if}] & \dfrac{A \vdash e_0 : u_0 \quad A \vdash e_1 : u_1 \quad A \vdash e_2 : u_2}{A \vdash \mathtt{if0}^l\, e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : u} & \text{if } u_0 \leq u_{\text{int}} \wedge u_1 \leq u \wedge u_2 \leq u
\end{array}$$

Fig. 1. The typing rules

$$\dfrac{\dfrac{\dfrac{\dfrac{A_{\mathsf{g}} \vdash \mathsf{g}^3 : u_{\mathsf{x}} \quad A_{\mathsf{g}} \vdash \mathsf{g}^4 : u'_{\mathsf{x}}}{A_{\mathsf{g}} \vdash \mathsf{g}^3 @_2 \mathsf{g}^4 : u'_{\mathsf{x}}} \quad A_{\mathsf{g}} \vdash 0^5 : u_{\text{int}}}{A_{\mathsf{g}} \vdash (\mathsf{g}^3 @_2 \mathsf{g}^4) @_1 0^5 : u_{\text{int}}}}{[\,] \vdash \lambda^6 \mathsf{g}.((\mathsf{g}^3 @_2 \mathsf{g}^4) @_1 0^5) : u_{\mathsf{g}}} \quad \dfrac{A_{\mathsf{x}} \vdash \mathtt{x}^7 : u_{\text{int}} \quad A'_{\mathsf{x}} \vdash \mathtt{x}^7 : u'_{\mathsf{x}}}{[\,] \vdash \lambda^8 \mathtt{x}.\mathtt{x}^7 : u_{\mathsf{x}}}}{[\,] \vdash (\lambda^6 \mathsf{g}.((\mathsf{g}^3 @_2 \mathsf{g}^4) @_1 0^5)) @_0 (\lambda^8 \mathtt{x}.\mathtt{x}^7) : u_{\text{int}}}$$

Fig. 2. A derivation D_{T_1} for the program P_1 from Example 1.

which provides a way of keeping track of the origin of a function type (cf. Sects. 1 and 5). Accordingly, the fun-*witness* w^λ of this inference is the pair $(q : t)$. Note that K may be empty in which case the body is not analyzed.

Example 2. For the program P_1 from Ex. 1, we can construct a typing T_1 as follows: $IT_{\mathsf{T}_1} = \{0, 1, 2\}$, $UT_{\mathsf{T}_1} = \{q_{\mathsf{x}}, q_{\mathsf{g}}\}$, and D_{T_1} is as in Fig. 2, where

$$\begin{array}{l}
u'_{\mathsf{x}} = \bigvee(q_{\mathsf{x}} : \bigwedge(\{1\} : u_{\text{int}} \to u_{\text{int}})) \\
u_{\mathsf{x}} = \bigvee(q_{\mathsf{x}} : \bigwedge(\{1\} : u_{\text{int}} \to u_{\text{int}}, \{2\} : u'_{\mathsf{x}} \to u'_{\mathsf{x}})) \\
u_{\mathsf{g}} = \bigvee(q_{\mathsf{g}} : \bigwedge(\{0\} : u_{\mathsf{x}} \to u_{\text{int}})) \\
A_{\mathsf{g}} = [\mathsf{g} \mapsto u_{\mathsf{x}}] \qquad A_{\mathsf{x}} = [\mathtt{x} \mapsto u_{\text{int}}] \qquad A'_{\mathsf{x}} = [\mathtt{x} \mapsto u'_{\mathsf{x}}]
\end{array}$$

Note that $u_{\mathsf{x}} \leq u'_{\mathsf{x}}$, and that $u_{\mathsf{x}}.q_{\mathsf{x}} \leq_\wedge \bigwedge(\{2\} : u'_{\mathsf{x}} \to u'_{\mathsf{x}})$ so that $\{q_{\mathsf{x}} \mapsto \{2\}\}$ is indeed an app-witness for the inference at the top left of Fig. 2.

The type system in Fig. 1 can be augmented with rules for `bind` and `close` such that the resulting system satisfies a subject reduction property. The soundness of the type system follows from subject reduction, since "stuck" expressions (such as $7 @ 9$) are not typable.

In a typing T for P, for each e in $\mathit{SubExpr}_P$ there may be several judgements for e in D_T, due to the multiple analyses performed by $[\mathsf{fun}]$. We assign to each judgement J for e in D_T an environment ke (its *address*) that for all applications of [fun] in the path from the root of D_T to J associates the bound variables with the branch taken. In D_{T_1} (Fig. 2), the judgement $A_{\mathtt{x}} \vdash \mathtt{x}^7 : u_{\text{int}}$ has address $[\mathtt{x} \mapsto 1]$ and the judgement $A'_{\mathtt{x}} \vdash \mathtt{x}^7 : u'_{\mathtt{x}}$ has address $[\mathtt{x} \mapsto 2]$.

The translation in Sect. 5 requires that a typing must be *uniform*, i.e., the following partial function A_T must be well-defined: $A_T(z, k) = u$ iff D_T contains a judgement of the form $A \vdash e : u'$ with address ke, where $ke(z) = k$ and $A(z) = u$. For T_1 we have, e.g., $A_{\mathsf{T}_1}(\mathtt{x}, 1) = u_{\text{int}}$ and $A_{\mathsf{T}_1}(\mathtt{x}, 2) = u'_{\mathtt{x}}$.

4 The Flow System

Our system for flow analysis has the form of a flow logic, in the style of N&N. A flow analysis F for program P is a tuple $(P, \mathit{Mem}_F, \mathcal{C}_F, \rho_F, \Phi_F)$, whose components are explained below (together with some auxiliary derived concepts).

Polyvariance is modeled by *mementoes*, where a memento ($m \in \mathit{Mem}_F$) represents a context for analyzing the body of a function. We shall assume that Mem_F is non-empty and *finite*; then all other entities occurring in F will also be finite. Each expression e is analyzed wrt. several different memento environments, where the entries of a memento environment ($me \in \mathit{MemEnv}_F$) take the form $[z \mapsto m]$ with m in Mem_F. Accordingly, a *flow configuration* ($\in \mathit{FlowConf}_F$) is a pair (e, me), where $FV(e) \subseteq dom(me)$.

The goal of the flow analysis is to associate a set of *flow values* to each configuration, where a flow value ($v \in \mathit{FlowVal}_F$) is either an integer Int or of the form (ac, M), where ac ($\in \mathit{AbsClos}_F$) is an *abstract closure* of the form (fn, me) with fn a function $\mu f.\lambda x.e$ and $FV(\mathit{fn}) \subseteq dom(me)$, and where $M \subseteq \mathit{Mem}_F$. The M component can be thought of a superset of the "sinks" of the abstract closure ac, i.e. the contexts in which it is going to be applied. Our flow values differ from N&N's in two respects: *(i)* they do not include the memento that corresponds to the point of definition; *(ii)* they do include the mementoes of use (the M component), in order to get a flow system that is almost isomorphic to the type system of Sect. 3. This extension does not make it harder to analyze an expression, since one might just let $M = \mathit{Mem}_F$ everywhere.

A *flow set* V ($\in \mathit{FlowSet}_F$) is a set of flow values, with the property that if $(ac, M_1) \in V$ and $(ac, M_2) \in V$ then $M_1 = M_2$. We define an ordering on $\mathit{FlowSet}_F$ by stipulating that $V_1 \leq_V V_2$ iff for all $v_1 \in V_1$ there exists $v_2 \in V_2$ such that $v_1 \leq_v v_2$, where the ordering $\leq_v$ on $\mathit{FlowVal}_F$ is defined by stipulating that Int $\leq_v$ Int and that $(ac, M_1) \leq_v (ac, M_2)$ iff $M_2 \subseteq M_1$. Note that if $V_1 \leq_V V_2$ then V_2 is obtained from V_1 by adding some "sources" and removing some "sinks" (in a sense moving along a "flow path" from a source to a sink), so in that respect the ordering is similar to the type ordering in [WDMT97].

Φ_F is a partial mapping from $(\mathit{Labs}_P \times \mathit{MemEnv}_F) \times \mathit{AbsClos}_F$ to $\mathcal{P}(\mathit{Mem}_F)$, where Labs_P is the set of labels occurring in P. Intuitively, if the abstract closure ac in the context me is applied to an expression with label l, then $\Phi_F((l, me), ac)$ denotes the actual sinks of ac.

$\mathcal{C}_F$ is a mapping from $\mathit{Labs}_P \times \mathit{MemEnv}_F$ to $(\mathit{FlowSet}_F)_\perp$. Intuitively, if $\mathcal{C}_F(l, me) = V$ $(\neq \perp)$ and $\mathcal{C}_F$ is valid (defined below) for the flow configuration (ue^l, me) then all semantic values that ue^l may evaluate to in a semantic environment approximated by me can be approximated by the set V. Similarly, $\rho_F(z, m)$ approximates the set of semantic values to which z may be bound when analyzed in memento m.

Unlike N&N, we distinguish between $\mathcal{C}_F(l, me)$ being the empty set and being $\perp$. The latter means that no flow configuration (ue^l, me) is "reachable", and so there is no need to analyze it. The relation $\leq_V$ on $\mathit{FlowSet}_F$ is lifted to a relation $\leq_V$ on $\mathit{FlowSet}_{F\perp}$.

Example 3. For the program P_1 from Ex. 1, a flow analysis F_1 with $\mathit{Mem}_{\mathsf{F}_1} = \{0, 1, 2\}$ is given below. We have named some entities (note that $v_{\mathtt{x}} \leq_v v'_{\mathtt{x}}$):

$$\begin{array}{lll} me_{\mathsf{g}} = [\mathsf{g} \mapsto 0] & ac_{\mathsf{g}} = (\lambda \mathsf{g}.\cdots, [\,]) & v_{\mathsf{g}} = (ac_{\mathsf{g}}, \{0\}) \\ me_{\mathtt{x1}} = [\mathtt{x} \mapsto 1] & ac_{\mathtt{x}} = (\lambda \mathtt{x}.\mathtt{x}^7, [\,]) & v'_{\mathtt{x}} = (ac_{\mathtt{x}}, \{1\}) \\ me_{\mathtt{x2}} = [\mathtt{x} \mapsto 2] & & v_{\mathtt{x}} = (ac_{\mathtt{x}}, \{1, 2\}) \end{array}$$

$\mathcal{C}_{\mathsf{F}_1}$ and ρ_{F_1} are given by the entries below (all other are $\perp$):

$$\begin{aligned} \{v_{\mathsf{g}}\} &= \mathcal{C}_{\mathsf{F}_1}(6, [\,]) \\ \{\mathrm{Int}\} &= \rho_{\mathsf{F}_1}(\mathtt{x}, 1) = \mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x1}}) = \mathcal{C}_{\mathsf{F}_1}(5, me_{\mathsf{g}}) = \mathcal{C}_{\mathsf{F}_1}(1, me_{\mathsf{g}}) = \mathcal{C}_{\mathsf{F}_1}(0, [\,]) \\ \{v'_{\mathtt{x}}\} &= \rho_{\mathsf{F}_1}(\mathtt{x}, 2) = \mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x2}}) = \mathcal{C}_{\mathsf{F}_1}(4, me_{\mathsf{g}}) = \mathcal{C}_{\mathsf{F}_1}(2, me_{\mathsf{g}}) \\ \{v_{\mathtt{x}}\} &= \rho_{\mathsf{F}_1}(\mathsf{g}, 0) = \mathcal{C}_{\mathsf{F}_1}(3, me_{\mathsf{g}}) = \mathcal{C}_{\mathsf{F}_1}(8, [\,]) \end{aligned}$$

Thus $(\mathsf{g}^3 \,@_2\, \mathsf{g}^4) \,@_1\, 0^5$ is analyzed with g bound to 0, and $\mathtt{x}^7$ is analyzed twice: with $\mathtt{x}$ bound to 1 and with $\mathtt{x}$ bound to 2. Accordingly, Φ_{F_1} is given by

$$\Phi_{\mathsf{F}_1}((8, [\,]), ac_{\mathsf{g}}) = \{0\},\ \Phi_{\mathsf{F}_1}((5, me_{\mathsf{g}}), ac_{\mathtt{x}}) = \{1\},\ \Phi_{\mathsf{F}_1}((4, me_{\mathsf{g}}), ac_{\mathtt{x}}) = \{2\}.$$

4.1 Validity

Of course, not all flow analyses give a correct description of the program being analyzed. To formulate a notion of validity, we define a predicate $F \models^{me} e$ (to be read: F analyzes e correctly wrt. the memento environment me), with $(e, me) \in \mathit{FlowConf}_F$. The predicate must satisfy the specification in Fig. 3, which gives rise to a monotone functional on the complete lattice $\mathcal{P}(\mathit{FlowConf}_F)$; following the convincing argument of N&N, we define $F \models^{me} e$ as the *greatest* fixed point of this functional so as to be able to cope with recursive functions.

In [*fun*], we deviate from N&N by recording me, rather than the restriction of me to $FV(\mu f.\lambda x.e_0)$. As in P&P, this facilitates the translations to and from types. In [*app*], the set M corresponds to P&P's notion of *cover*, which in turn is needed to model the "cartesian product" algorithm of [Age95]. In N&N's framework, M is always a singleton $\{m\}$; in that case the condition "$\forall v \in \mathcal{C}_F(l_2, me).\ \ldots$ " amounts to the simpler "$\mathcal{C}_F(l_2, me) \leq_V \rho_F(x, m)$".

By structural induction in ue^l we see that if $F \models^{me} ue^l$ then $\mathcal{C}_F(l, me) \neq \perp$. We would also like the converse implication to hold:

$[var]$ $\quad F \models^{me} z^l$ iff $\bot \neq \rho_F(z, me(z)) \ \leq_V \ \mathcal{C}_F(l, me)$

$[fun]$ $\quad F \models^{me} \mu f.\lambda^l x.e_0$ iff $\{((\mu f.\lambda x.e_0, me), \mathit{Mem}_F)\} \leq_V \mathcal{C}_F(l, me)$

$[app]$ $\quad F \models^{me} {ue_1}^{l_1} \ @_l \ {ue_2}^{l_2}$ iff

$$\begin{array}{l}
\mathcal{C}_F(l, me) \neq \bot \wedge F \models^{me} {ue_1}^{l_1} \wedge F \models^{me} {ue_2}^{l_2} \wedge \\
\forall (ac_0, M_0) \in \mathcal{C}_F(l_1, me) \\
\qquad \text{let } M = \Phi_F((l_2, me), ac_0) \text{ and } (\mu f.\lambda x.{ue_0}^{l_0}, me_0) = ac_0 \text{ in} \\
\qquad M \subseteq M_0 \wedge \forall v \in \mathcal{C}_F(l_2, me).\, \exists m \in M.\, \{v\} \leq_V \rho_F(x, m) \wedge \\
\qquad \forall m \in M\colon F \models^{me_0[f,x \mapsto m]} {ue_0}^{l_0} \wedge \\
\qquad\qquad \mathcal{C}_F(l_0, me_0[f, x \mapsto m]) \leq_V \mathcal{C}_F(l, me) \wedge \\
\qquad\qquad \rho_F(x, m) \neq \bot \wedge \{(ac_0, \mathit{Mem}_F)\} \leq_V \rho_F(f, m)
\end{array}$$

$[con]$ $\quad F \models^{me} c^l$ iff $\mathrm{Int} \in \mathcal{C}_F(l, me)$

$[suc]$ $\quad F \models^{me} \mathtt{succ}^l \ e_1$ iff $F \models^{me} e_1 \wedge \mathrm{Int} \in \mathcal{C}_F(l, me)$

$[if]$ $\quad F \models^{me} \mathtt{if0}^l \ e_0 \ \mathtt{then} \ {ue_1}^{l_1} \ \mathtt{else} \ {ue_2}^{l_2}$ iff

$$\begin{array}{l}
F \models^{me} e_0 \wedge F \models^{me} {ue_1}^{l_1} \wedge F \models^{me} {ue_2}^{l_2} \wedge \\
\mathcal{C}_F(l_1, me) \leq_V \mathcal{C}_F(l, me) \wedge \mathcal{C}_F(l_2, me) \leq_V \mathcal{C}_F(l, me)
\end{array}$$

Fig. 3. The flow logic

Definition 1. *Let a flow analysis F for P be given. We say that F is valid iff* (i) $F \models^{[]} P$; (ii) *whenever* $e = ue^l \in \mathit{SubExpr}_P$ *with* $(e, me) \in \mathit{FlowConf}_F$ *and* $\mathcal{C}_F(l, me) \neq \bot$ *then* $F \models^{me} e$.

Using techniques as in N&N, we can augment Fig. 3 with rules for `bind` and `close` and then prove a subject reduction property for flows which for closed E reads: if E reduces to E' in one evaluation step and $F \models^{[]} E$ then $F \models^{[]} E'$.

So far, even for badly behaved programs like $P = 7 \,@\, 9$ it is possible (just as in N&N) to find a F for P such that F is valid. Since our type system rejects such programs, we would like to filter them out:

Definition 2. *Let a flow analysis F for P be given. We say that F is safe iff for all ue^l in $\mathit{SubExpr}_P$ and for all me it holds:* (i) *if* $ue = {ue_1}^{l_1} \,@\, e_2$ *then* $\mathit{Int} \notin \mathcal{C}_F(l_1, me)$; (ii) *if* $ue = \mathtt{succ}\ {ue_1}^{l_1}$ *then* $v \in \mathcal{C}_F(l_1, me)$ *implies* $v = \mathit{Int}$; (iii) *if* $ue = \mathtt{if0}\ {ue_0}^{l_0}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ *then* $v \in \mathcal{C}_F(l_0, me)$ *implies* $v = \mathit{Int}$.

Example 4. Referring back to Example 3, it clearly holds that F_1 is safe, and it is easy (though a little cumbersome) to verify that F_1 is valid.

4.2 Taxonomy of Flow Analyses

Two common categories of flow analyses are the "call-string based" (e.g., [Shi91]) and the "argument-based" (e.g., [Sch95,Age95]). Our *descriptive* framework can model both approaches (which can be "mixed", as in [NN99]).

A flow analysis F for P such that F is valid is in $\mathit{CallString}_\beta^P$, where β is a mapping from $\mathit{Labs}_P \times \mathit{MemEnv}_F$ into Mem_F, iff whenever $\Phi_F((l_2, me), ac)$ is defined it equals $\{\beta(l, me)\}$ where l is such that[4] $e_1 \mathbin{@_l} ue_2{}^{l_2} \in \mathit{SubExpr}_P$. All k-CFA analyses fit into this category: for 0-CFA we take $\mathit{Mem}_F = \{\bullet\}$ and $\beta(l, me) = \bullet$; for 1-CFA we take $\mathit{Mem}_F = \mathit{Labs}_P$ and $\beta(l, me) = l$; and for 2-CFA (the generalization to $k > 2$ is immediate) we take $\mathit{Mem}_F = \mathit{Labs}_P \cup (\mathit{Labs}_P \times \mathit{Labs}_P)$ and define $\beta(l, me)$ as follows: let it be l if $me = [\,]$, and let it be (l, l_1) if me takes the form $me'[z \mapsto m]$ with m either l_1 or (l_1, l_2).

A flow analysis F for P such that F is valid is in $\mathit{ArgBased}_\alpha^P$ iff for all non-recursive variables x and mementoes m it holds that whenever $\rho_F(x, m) \neq \bot$ then $\epsilon_V(\rho_F(x, m)) = \alpha(m)$ where ϵ_V removes the M component of a flow value. For this kind of analysis, a memento m essentially denotes a set of abstract closures. To more precisely capture specific argument-based analyses, such as [Age95] or the type-directed approach of [JWW97], we may impose further demands on α.

Example 5. The flow analysis F_1 is a 1-CFA and also in $\mathit{ArgBased}_\alpha^{\mathsf{P}_1}$, with $\alpha(0) = \alpha(2) = \{ac_{\mathrm{x}}\}$ and $\alpha(1) = \{\mathrm{Int}\}$.

Given a program P, it turns out that for all β the class $\mathit{CallString}_\beta^P$, and for certain kinds of α also the class $\mathit{ArgBased}_\alpha^P$, contains a least (i.e., most precise) flow analysis; here the ordering on flow analyses is defined pointwise[5] on $\mathcal{C}_F$, ρ_F and Φ_F. This is much as in N&N where for all total and deterministic "instantiators" the corresponding class of analyses contains a least element, something we cannot hope for since we allow Φ_F to return a non-singleton.

4.3 Reachability

For a flow analysis F, some entries may be garbage. To see an example of this, suppose that $\mu f.\lambda x.ue^l$ in $\mathit{SubExpr}_P$, and suppose that $\rho_F(x, m) = \bot$ for all $m \in \mathit{Mem}_F$. From this we infer that the above function is never called, so for all me the value of $\mathcal{C}_F(l, me)$ is uninteresting. It may therefore be replaced by $\bot$, something which is in fact achieved by the roundtrip described in Sect. 7.1.

To formalize a notion of reachability we introduce a set Reach_P^F that is intended to encompass[6] all entries of $\mathcal{C}_F$ and ρ_F that are "reachable" from the root of P. Let $\mathit{Analyzes}_m^F(\mu f.\lambda x.ue_0{}^{l_0}, me)$ be a shorthand for $\mathcal{C}_F(l_0, me[f, x \mapsto m]) \neq \bot$ and $\rho_F(x, m) \neq \bot$ and $\{((\mu f.\lambda x.ue_0{}^{l_0}, me), \mathit{Mem}_F)\} \leq_V \rho_F(f, m)$. We define Reach_P^F as the least set satisfying:

$$[\mathrm{prg}] \quad (P, [\,]) \in \mathit{Reach}_P^F$$

$$[\mathrm{fun}] \quad \Big((\mu f.\lambda^l x.ue_0{}^{l_0}, me) \in \mathit{Reach}_P^F \;\wedge\; \mathit{Analyzes}_m^F(\mu f.\lambda x.ue_0{}^{l_0}, me)\Big) \Rightarrow \\ \Big((ue_0{}^{l_0}, me[f, x \mapsto m]) \in \mathit{Reach}_P^F \;\wedge\; (x, m) \in \mathit{Reach}_P^F \;\wedge\; (f, m) \in \mathit{Reach}_P^F\Big)$$

[4] It is tempting to write "$\Phi_F((l, me), ac_0)$" in Fig. 3 (thus replacing l_2 by l), but then subject reduction for flows would not hold.

[5] Unlike [JWW97], we do not compare analyses with different sets of mementoes.

[6] This is somewhat similar to the reachability predicate of [GNN97].

[app] $(e_1 \,@_l\, e_2, me) \in \mathit{Reach}_P^F \Rightarrow (e_1, me) \in \mathit{Reach}_P^F \wedge (e_2, me) \in \mathit{Reach}_P^F$

[suc] $(\mathtt{succ}^l\ e_1, me) \in \mathit{Reach}_P^F \Rightarrow (e_1, me) \in \mathit{Reach}_P^F$

[if] $(\mathtt{if0}^l\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, me) \in \mathit{Reach}_P^F \Rightarrow$
$(e_0, me) \in \mathit{Reach}_P^F \wedge (e_1, me) \in \mathit{Reach}_P^F \wedge (e_2, me) \in \mathit{Reach}_P^F$

Example 6. It is easy to verify that for $ue^l \in \mathit{SubExpr}_{\mathsf{P}_1}$ it holds that $\mathcal{C}_{\mathsf{F}_1}(l, me) \neq \bot$ iff $(ue^l, me) \in \mathit{Reach}_{\mathsf{P}_1}^{\mathsf{F}_1}$, and that $\rho_{\mathsf{F}_1}(z, m) \neq \bot$ iff $(z, m) \in \mathit{Reach}_{\mathsf{P}_1}^{\mathsf{F}_1}$.

Lemma 2. *Let F be a flow analysis for P such that F is valid. If $(ue^l, me) \in \mathit{Reach}_P^F$ then* (i) *$\mathcal{C}_F(l, me) \neq \bot$ and* (ii) *whenever $(z \mapsto m) \in me$ then $(z, m) \in \mathit{Reach}_P^F$ holds. Also, if $(z, m) \in \mathit{Reach}_P^F$ then $\rho_F(z, m) \neq \bot$.*

5 Translating Types to Flows

Let a uniform typing T for a program P be given. We now demonstrate how to construct a corresponding flow analysis $F = \mathcal{F}(T)$ such that F is valid and safe. First define Mem_F as IT_T; note that then an address can serve as a memento environment. Next we define a function $\mathcal{F}_T$ that translates from UTyp_T, that is the union types that can be built using IT_T and UT_T, into $\mathit{FlowSet}_F$:

$\mathcal{F}_T(\bigvee_{i \in I}\{q_i : t_i\}) =$
$\{((\mu f.\lambda x.e, me), M) \mid \exists i \in I$ with $M = \mathit{dom}(t_i)$:
a judgement for $\mu f.\lambda^l x.e$ occurs in D_T with address me
and is justified by $[\mathsf{fun}]^{(q_i : t)}$ where $t \leq_\wedge t_i\}$
$\cup$ (if $\exists i.$ such that $q_i = q_{\text{int}}$ then $\{\text{Int}\}$ else $\emptyset$)

The idea behind the translation is that $\mathcal{F}_T(u)$ should contain all the closures that are "sources" of elementary types in u; it is easy to trace such closures thanks to the presence of U-tags. The condition $t \leq_\wedge t_i$ is needed as a "sanity check", quite similar to the "trimming" performed in [Hei95], to guard against the possibility that two unrelated entities in D_T incidentally have used the same U-tag q_i. As the types of P&P do not contain fun-witnesses, their translation has to rely solely on this sanity check (at the cost of precision, cf. Sect. 1).

Lemma 3. *The function $\mathcal{F}_T$ is monotone.*

Definition 3. *With T a typing for P, the flow analysis $F = \mathcal{F}(T)$ is given by $(P, \mathit{IT}_T, \mathcal{C}_F, \rho_F, \Phi_F)$, where $\mathcal{C}_F$, ρ_F, and Φ_F are defined below:*

$\mathcal{C}_F(l, me) = \mathcal{F}_T(u)$ *iff D_T contains a judgement $A \vdash ue^l : u$ with address me*

$\rho_F(z, m) = \mathcal{F}_T(u)$ *iff $u = A_T(z, m)$*

$\Phi_F((l_2, me), (\mu f.\lambda x.e_0, me')) = M$ *iff there exists q such that D_T contains*
a judgement for $\mu f.\lambda x.e_0$ at me' derived by $[\mathsf{fun}]^{(q:t)}$,
a judgement for $e_1 \,@\, ue_2{}^{l_2}$ at me derived by $[\mathsf{app}]^{w^@}$ where $w^@(q) = M$.

Example 7. With terminology as in Examples 2 and 3, it is easy to check that $\mathcal{F}_{\mathsf{T}_1}(u'_{\mathtt{x}}) = \{v'_{\mathtt{x}}\}$ and that $\mathcal{F}_{\mathsf{T}_1}(u_{\mathtt{x}}) = \{v_{\mathtt{x}}\}$, and that $\mathsf{F}_1 = \mathcal{F}(\mathsf{T}_1)$.

We have the following result, where the proof that F is valid is by coinduction.

Theorem 1. *With T a uniform typing for P, for $F = \mathcal{F}(T)$ it holds that*

- *F is valid and safe*
- *$(ue^l, me) \in \mathit{Reach}_P^F$ iff $\mathcal{C}_F(l, me) \neq \bot$ (for $ue \in \mathit{SubExpr}_P$)*
- *$(z, m) \in \mathit{Reach}_P^F$ iff $\rho_F(z, m) \neq \bot$*

6 Translating Flows to Types

Let a flow analysis F for a program P be given, and assume that F is valid and safe. We now demonstrate how to construct a corresponding uniform typing $T = \mathcal{T}(F)$. First we define IT_T as Mem_F and UT_T as $\mathit{AbsClos}_F \cup \{q_{\text{int}}\}$. Next we define a function $\mathcal{T}_F$ that translates from $\mathit{FlowSet}_F$ into UTyp_T; inspired by P&P (though the setting is somewhat different) we stipulate:

$\mathcal{T}_F(V) = \bigvee_{v \in V} \{q_v : t_v\}$ where
  if $v = \text{Int}$ then $q_v = q_{\text{int}}$ and $t_v = \mathtt{int}$
  if $v = (ac, M)$ with $ac = (\mu f.\lambda x.{e_0}^{l_0}, me)$
    then $q_v = ac$
    and $t_v = \bigwedge_{m \in M_0} \{\{m\} : \mathcal{T}_F(\rho_F(x, m)) \to \mathcal{T}_F(\mathcal{C}_F(l_0, me[f, x \mapsto m]))\}$
      where $M_0 = \{m \in M \mid \mathit{Analyzes}_m^F(ac)\}$.

The above definition determines a unique union type $\mathcal{T}_F(V)$, since recursion is "beneath a constructor" and since $\mathit{FlowSet}_F$ is finite (ensuring regularity).

Example 8. With terminology as in Examples 2 and 3, it is easy to see—provided that $q_{\mathtt{x}}$ is considered another name for $ac_{\mathtt{x}}$—first that $\mathcal{T}_{\mathsf{F}_1}(\{v'_{\mathtt{x}}\}) = u'_{\mathtt{x}}$, and then that $\mathcal{T}_{\mathsf{F}_1}(\{v_{\mathtt{x}}\}) = u_{\mathtt{x}}$ since $\mathcal{T}_{\mathsf{F}_1}(\{v_{\mathtt{x}}\}).q_{\mathtt{x}}$ can be found as

$$\begin{aligned}
&\bigwedge(\{1\} : \mathcal{T}_{\mathsf{F}_1}(\rho_{\mathsf{F}_1}(\mathtt{x}, 1)) \to \mathcal{T}_{\mathsf{F}_1}(\mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x}1})), \{2\} : \mathcal{T}_{\mathsf{F}_1}(\rho_{\mathsf{F}_1}(\mathtt{x}, 2)) \to \mathcal{T}_{\mathsf{F}_1}(\mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x}2})))\\
=\ &\bigwedge(\{1\} : \mathcal{T}_{\mathsf{F}_1}(\{\text{Int}\}) \to \mathcal{T}_{\mathsf{F}_1}(\{\text{Int}\}), \{2\} : \mathcal{T}_{\mathsf{F}_1}(\{v'_{\mathtt{x}}\}) \to \mathcal{T}_{\mathsf{F}_1}(\{v'_{\mathtt{x}}\}))\\
=\ &\bigwedge(\{1\} : u_{\text{int}} \to u_{\text{int}}, \{2\} : u'_{\mathtt{x}} \to u'_{\mathtt{x}}).
\end{aligned}$$

Note that without the M component in a flow value (ac, M), $v_{\mathtt{x}}$ would equal $v'_{\mathtt{x}}$ causing $\mathcal{T}_{\mathsf{F}_1}(\{v_{\mathtt{x}}\})$ to be an infinite type (as in P&P).

Lemma 4. *The function $\mathcal{T}_F$ is monotone.*

For z and m such that $(z, m) \in \mathit{Reach}_P^F$, we define $\mathcal{T}_F^\rho(z, m)$ as $\mathcal{T}_F(\rho_F(z, m))$ (by Lemma 2 this is well-defined). And for $e = ue^l$ and me such that $(e, me) \in \mathit{Reach}_P^F$, we construct a judgement $\mathcal{T}_F^J(e, me)$ as

$$\mathcal{T}_F^A(me) \vdash e : \mathcal{T}_F(\mathcal{C}_F(l, me))$$

where $\mathcal{T}_F^A(me)$ is defined recursively by $\mathcal{T}_F^A([\,]) = [\,]$ and $\mathcal{T}_F^A(me[z \mapsto m]) = \mathcal{T}_F^A(me)[z \mapsto \mathcal{T}_F^\rho(z, m)]$ (by Lemma 2 also this is well-defined).

Definition 4. *With F a flow analysis for P, the typing $T = \mathcal{T}(F)$ is given by $(P, Mem_F, AbsClos_F \cup \{q_{\text{int}}\}, D_T)$, where D_T is defined by stipulating that whenever (e, me) is in $Reach_P^F$ then D_T contains $\mathcal{T}_F^J(e, me)$, and that $\mathcal{T}_F^J(e', me')$ is a premise of $\mathcal{T}_F^J(e, me)$ iff $(e, me) \in Reach_P^F$ is among the immediate conditions (cf. the definition of $Reach_P^F$) for $(e', me') \in Reach_P^F$.*

Example 9. It is easy to check that $\mathsf{T}_1 = \mathcal{T}(\mathsf{F}_1)$, modulo renaming of the U-tags.

Clearly D_T is a tree-formed derivation, and $\mathcal{T}_F^J(e, me)$ has address me in D_T. We must of course also prove that all judgements in D_T are in fact derivable from their premises using the inference rules in Fig. 1:

Theorem 2. *If F is valid and safe then $T = \mathcal{T}(F)$ as constructed by Definition 4 is a uniform typing for P. The derivation D_T has the following properties:*

- *if D_T contains at address me a judgement for $\mu f.\lambda x.e$, it is derived using $[\mathsf{fun}]^{w^\lambda}$ where $w^\lambda = (ac : (\mathcal{T}_F(\{(ac, Mem_F)\})).ac)$ with $ac = (\mu f.\lambda x.e, me)$;*
- *if D_T contains at address me a judgement for $e_1 \,@\, u{e_2}^{l_2}$ with the leftmost premise of the form $A \vdash e_1 : u_1$, then it is derived using $[\mathsf{app}]^{w^@}$ where for all $q \in dom(u_1)$ it holds that $w^@(q) = \Phi_F((l_2, me), q)$.*

7 Round Trips

Next consider the "round-trip" translations $\mathcal{F} \circ \mathcal{T}$ (from flows to types and back) and $\mathcal{T} \circ \mathcal{F}$ (from types to flows and back). Both roundtrips are idempotent[7]: they act as the identity on "canonical" elements, and otherwise "canonicalize".

Example 10. Exs. 7 and 9 show that $\mathcal{F} \circ \mathcal{T}$ is the identity on F_1 and that $\mathcal{T} \circ \mathcal{F}$ is the identity (modulo renaming of U-tags) on T_1. In particular $\mathcal{T} \circ \mathcal{F}$ does not necessarily introduce infinite types, thus solving an open problem in P&P.

7.1 Round Trips from the Flow World

$\mathcal{F} \circ \mathcal{T}$ filters out everything not reachable, and acts as the identity ever after.

Theorem 3. *Assume that F is valid and safe for a program P, and let $F' = \mathcal{F}(\mathcal{T}(F))$. Then F' is valid and safe for P with $Mem_{F'} = Mem_F$, $Reach_P^{F'} = Reach_P^F$, and $\mathcal{C}_{F'}(l, me) \neq \bot$ iff $\Big(\mathcal{C}_F(l, me) \neq \bot$ and $(ue^l, me) \in Reach_P^F\Big)$ in which case $\mathcal{C}_{F'}(l, me) = \text{filter}_P^F(\mathcal{C}_F(l, me))$ where $\text{filter}_P^F(V)$ is given by*

$$\begin{aligned}&\{(ac, M') \mid (ac, M) \in V \text{ and } (\mu f.\lambda x.e_0, me_0) \in Reach_P^F \text{ where}\\ &\qquad ac = (\mu f.\lambda x.e_0, me_0) \text{ and } M' = \{m \in M \mid (e_0, me_0[f, x \mapsto m]) \in Reach_P^F\}\\ &\cup(\text{if } Int \in V \text{ then } \{Int\} \text{ else } \emptyset).\end{aligned}$$

Finally, if $F'' = \mathcal{F}(\mathcal{T}(F'))$ then $F'' = F'$.

[7] However, $\mathcal{T}(\mathcal{F}(\mathcal{T}(F))) = \mathcal{T}(F)$ does in general not hold.

Clearly everything not reachable may be considered "junk". However, simple examples demonstrate that some junk is reachable and is hence not removed by $\mathcal{F} \circ \mathcal{T}$. That our flow/type correspondence can faithfully encode such imprecisions illustrates the power of our framework.

7.2 Round Trips from the Type World

The canonical typings are the ones that are *strongly consistent*:

Definition 5. *A typing T is strongly consistent iff for all u that occur in D_T and for all $q \in dom(u)$ with $q \neq q_{\text{int}}$ the following holds: D_T contains exactly one judgement derived by an application of $[\mathsf{fun}]^{w^\lambda}$ with w^λ taking the form $(q : t)$, and this t satisfies $t \leq^c_\wedge u.q$. Here $\leq^c_\wedge$ is a subrelation of $\leq_\wedge$, defined by stipulating that $\mathtt{int} \leq^c_\wedge \mathtt{int}$ and that $\bigwedge_{i \in I}\{K_i : u_i \to u'_i\} \leq^c_\wedge \bigwedge_{i \in I_0}\{K_i : u_i \to u'_i\}$ iff $I_0 \subseteq I$.*

Theorem 4. *Assume that T is a uniform typing for a program P, and let $T' = \mathcal{T}(\mathcal{F}(T))$. Then T' is a uniform typing for P with $IT_{T'} = IT_T$, and*
- *$D_{T'}$ contains a judgement for e with address ke iff D_T contains a judgement for e with address ke (i.e., the two derivations have the same shape);*
- *$D_{T'}$ is strongly consistent;*
- *if D_T is strongly consistent then $D_{T'} = D_T$ (modulo renaming of U-tags).*

Example 11. Let T be the typing[8] of the motivating example put forward in Sect. 1. Then T is not strongly consistent, but $T' = \mathcal{T}(\mathcal{F}(T))$ is: the two fun-witnesses occurring in $D_{T'}$ are of the form $(q_{\mathtt{x}} : u_{\text{int}} \to u_{\text{int}})$ and $(q_{\mathtt{y}} : u_{\text{int}} \to u_{\text{int}})$. Nevertheless, T' is still imprecise: both function abstractions are assigned the union type $\bigvee(q_{\mathtt{x}} : u_{\text{int}} \to u_{\text{int}}, q_{\mathtt{y}} : u_{\text{int}} \to u_{\text{int}})$.

8 Discussion

Our flow system follows the lines of N&N, generalizing some features while omitting others (such as polymorphic splitting [WJ98], left for future work). That it has substantial descriptive power is indicated by the fact that it encompasses both argument-based and call-string based polyvariance. In particular, the flow analysis framework of P&P can be encoded into our framework. Unlike P&P, our flow logic has a subject reduction property, inherited from the N&N approach.

The generality of our type system is less clear. The annotation with tags gives rise to intersection and union types that are not associative, commutative, or idempotent (ACI). This stands in contrast to the ACI types of P&P, but is similar to the non-ACI intersection and union types of CIL, the intermediate language of an experimental compiler that integrates flow information into the type system [WDMT97,DMTW97]. Indeed, a key motivation of this work was to formalize the encoding of various flow analyses in the CIL type system. Developing a translation between the the type system of this paper and CIL is our next goal.

[8] We convert it to our framework by substituting u_{int} for $\mathtt{int}$ and by substituting $\bigvee(q_\bullet : \bigwedge(\{\bullet\} : u_{\text{int}} \to u_{\text{int}}))$ for $\mathtt{int} \to \mathtt{int}$.

References

[AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Prog. Langs. and Systs.*, 15(4):575–631, 1993.

[Age95] O. Agesen. The cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, vol. 952, pp. 2–26. Springer-Verlag, 1995.

[AT00] T. Amtoft and F. Turbak. Faithful translations between polyvariant flows and polymorphic types. Technical Report BUCS-TR-2000-01, Comp. Sci. Dept., Boston Univ., 2000.

[Ban97] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 [ICFP97].

[DMTW97] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [ICFP97], pp. 11–24.

[GNN97] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In ICFP '97 [ICFP97], pp. 38–51.

[Hei95] N. Heintze. Control-flow analysis and type systems. In SAS '95 [SAS95], pp. 189–206.

[ICFP97] *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

[JW95] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 393–407, 1995.

[JWW97] S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, vol. 1302 of *LNCS*. Springer-Verlag, 1997.

[NN97] F. Nielson and H. R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pp. 332–345, 1997.

[NN99] F. Nielson and H. R. Nielson. Interprocedural control flow analysis. In *Proc. European Symp. on Programming*, vol. 1576 of *LNCS*, pp. 20–39. Springer-Verlag, 1999.

[PO95] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Trans. on Prog. Langs. and Systs.*, 17(4):576–599, 1995.

[PP98] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, pp. 197–208, 1998. Superseded by [PP99].

[PP99] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. A substantially revised version of [PP98]. Available at http://www.cs.purdue.edu/homes/palsberg/paper /popl98.ps.gz., Feb. 1999.

[SAS95] *Proc. 2nd Int'l Static Analysis Symp.*, vol. 983 of *LNCS*, 1995.

[Sch95] D. Schmidt. Natural-semantics-based abstract interpretation. In SAS '95 [SAS95], pp. 1–18.

[Shi91] O. Shivers. *Control Flow Analysis of Higher Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[WDMT97] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pp. 757–771, 1997.

[WJ98] A. Wright and S. Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. on Prog. Langs. and Systs.*, 20:166–207, 1998.