Higher-Order Value Flow Graphs

Christian Mossin

DIKU, University of Copenhagen**

Abstract. The concepts of value- and control-flow graphs are important for program analysis of imperative programs. An imperative value flow graph can be constructed by a single pass over the program text. No similar concepts exist for higher-order languages: we propose a method for constructing value flow graphs for typed higher-order functional languages. A higher-order value flow graph is constructed by a single pass over an explicitly typed program. By using standard methods, single source and single use value flow problems can be answered in linear time and all sources-all uses can be answered in quadratic time (in the size of the flow graph, which is equivalent to the size of the explicitly typed program). On simply typed programs, the precision of the resulting analysis is equivalent to closure analysis [10,11,8]. In practice, it is a reasonable assumption that typed programs are only bigger than their untyped equivalent by a constant factor, hence this is an asymptotic improvement over previous algorithms.
We extend the analysis to handle polymorphism, sum types and recursive types. As a consequence, the analysis can handle (explicit) dynamically typed programs. The analysis is polyvariant for polymorphic definitions.

Keywords: program analysis, type system, efficiency, polymorphism, recursive types, polyvariance.

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.

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.

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

We present a flow analysis for typed, higher-order functional languages. The analysis constructs a value flow graph which is linear in the size of the explicitly typed program. Single queries (single-source or single-sink data-flow) can be performed on this graph by standard reachability algorithms in linear time and, similarly, full flow information can be obtained in quadratic time.

On simply typed programs our analysis is equivalent in strength to closure analysis [10,11] and the constraint based analysis of Palsberg [8]. Since explicitly typed programs are typically only a constant bigger than the underlying untyped program, this gives (under the assumption that all types are bounded by a constant) an asymptotic improvement over previously published algorithms.

Independently of this work, Heintze and McAllester [4] developed a constraint based analysis with the same properties as our (i.e. linear time single query flow analysis of simply typed programs) — a careful examination reveals that the analyses are indeed similar. The presentations, however, are fundamentally different. In particular, Heintze and McAllester do not develop an explicit flow graph and the concept of types only occurs in the complexity argument. In our approach, types are a integral part of the analysed program. This makes our analysis easier to extend: in this paper we show how to deal with *polymorphism* (Heintze and McAllester give an ad hoc solution basically unfolding all polymorphic definitions), *recursive* types (Heintze and McAllester shows how lists can be handled, but it is not clear how to generalise to arbitrary recursive types) and *sum* types (not considered beyond lists by Heintze and McAllester).

We add polymorphism, sums and recursive types by considering these constructs as new value/consumer pairs: abstraction/instantiation, injection/projection and fold/unfold. Being able to handle languages with sums and recursive types allows us to specify the analysis for languages with an explicit *dynamic* type system — thus making the analysis applicable for untyped as well as typed languages.

The analysis is polyvariant for polymorphic definitions: monomorphic program analysis lose precision at function definitions by mixing information from different call-sites. Our analysis avoids this for arguments of polymorphic type.

2 Language

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

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

We present the language using the type system of figure 1. In order to refer to subexpression *occurrences*, we assume that terms are *labelled*. 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).

The semantics of the language is given by the reduction rules in figure 2. As usual we write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. We assume for

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

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

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

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

Fig. 1. Type System

Contexts:

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

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. 2. Semantics

all expressions that bound and free variables are distinct, and that this property is preserved (by α-conversion) during reduction.

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

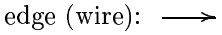
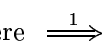
3 Flow Graphs

A *typed flow graph* for a type derivation $\mathcal{T}$ for an expression e is a graph (V, E) where V is a set of nodes and E is a set of edges.[1]. Each judgement $A \vdash e' : t$ in

[1] The reader might want to think of graphs as graphical representation of constraint sets: nodes n as variable and edges from n_1 to n_2 as constraints $n_1 \leq n_2$

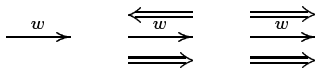
$\mathcal{T}$ is represented by a set of nodes: one node for each constructor (Bool, $\times$, $\rightarrow$) in t. The node associated with the top type constructor of t is named according to e while the rest are anonymous (but still conceptually associated with this named node). Collections of nodes associated with different judgements are called *multi-nodes* (or just m-nodes) and are connected by collections of edges called *cables* which intuitively carry values of the appropriate type. It is convenient to think of such an *m-node* as a parallel "plug". We will use N for m-nodes and n for single nodes. Each variable (bound or free) in the analysed expression e will give rise to *one* variable m-node and each *occurrence* of a variable gives rise to a *box* m-node. Every other subexpression e' of e will give rise to a *syntax* m-node and a *box* m-node. The latter is referred to as the *root* of the graph associated with e' and represents the result of evaluating e'.

The set of edges between two m-nodes form a *cable*. To be precise, we define a t-cable as follows:

1. A Bool-cable is a single edge (wire): 
2. A $(t \rightarrow t')$-cable is  where  is a t-cable, is its flipped version and is a t'-cable.
3. A $(t \times t')$-cable is  where  is a t-cable and is a t'-cable.

By "flipped" we mean inverting the direction of all wires in the cable, but not changing the top to bottom order of wires. We will use w for wires (edges) in E.

Edges are also considered paths (of length one). Composition of paths p_1 and p_2 is written $p_1 \cdot p_2$. If c is one of the following cables



the edge w is called the *carrier* of c. A path $w \cdot p \cdot w'$ is a *def-use* path if it starts from a data m-node, ends at a consumer m-node and w and w' are carriers. If w is an edge in a cable c, we say that it is a *forward* edge if it has the same orientation as the carrier of c, and a *backward* edge if it has the opposite orientation.

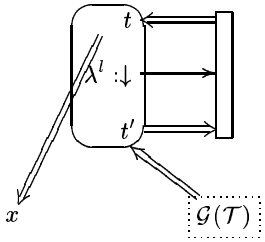
3.1 Simple Types

Figure 3 defines a function $\mathcal{G}$ from type derivations to typed flow graphs. Each right-hand side of the definition has a root m-node which is the m-node to be connected at recursive calls. Note that each data m-node generates a new carrier starting at the m-node and connects the sub-cables, while a consumer m-node terminates a carrier (and connects sub-cables). Furthermore, note that whenever two cables are connected, they have the same type.

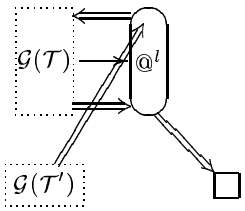
– The variable case constructs a root m-node and connects the (unique) variable m-node to the root.

$\mathcal{G}(\dfrac{}{A, x : t \vdash x : t}) \quad = \quad x \Longrightarrow \square \qquad$ where $\Longrightarrow$ is a t-cable

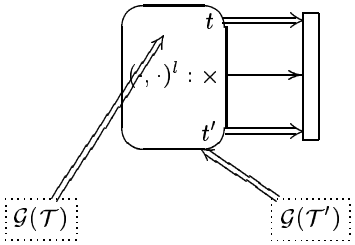
$\mathcal{G}(\dfrac{\mathcal{T}}{A \vdash \lambda^l x.e : t \to t'}) =$



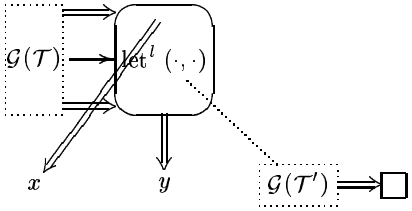
$\mathcal{G}(\dfrac{\mathcal{T} \quad \mathcal{T}'}{A \vdash e@^l e' : t}) =$

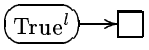


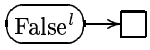
$\mathcal{G}(\dfrac{\mathcal{T} \quad \mathcal{T}'}{A \vdash (e, e')^l : t \times t'}) =$



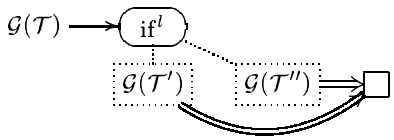
$\mathcal{G}(\dfrac{\mathcal{T} \quad \mathcal{T}'}{A \vdash \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : t''}) =$



$\mathcal{G}(\dfrac{}{A \vdash \text{True}^l : \text{Bool}}) =$ 

$\mathcal{G}(\dfrac{}{A \vdash \text{False}^l : \text{Bool}}) =$ 

$\mathcal{G}(\dfrac{\mathcal{T} \quad \mathcal{T}' \quad \mathcal{T}''}{A \vdash \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : t}) =$



$\mathcal{G}(\dfrac{\mathcal{T}}{A \vdash \text{fix}^l x.e : t}) =$

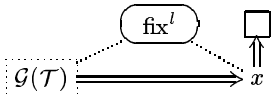


Fig. 3. Flow graphs for simply typed lambda calculus

- The case for $\mathcal{G}(\frac{\mathcal{T}}{A \vdash \lambda^l x.e : t \to t'})$ constructs a λ^l value m-node. A $t \to t'$ cable (call it c) leaves this m-node towards the root m-node of the graph indicating that the lambda itself is the result of the expression. A t cable connecting to the argument sub-cable of c goes towards the unique variable m-node for the bound variable. Similarly, a cable leaving the root of the graph for e connects to the result sub-cable of c.
  In the case for $e@^l e'$ a $t \to t'$ cable c enters a $@^l$ consumer m-node from the root of $\mathcal{G}(\mathcal{T})$ (where $\mathcal{T}$ is the derivation for e and $\mathcal{T}'$ is the derivation for e' — similar in the later cases). The root of $\mathcal{G}(\mathcal{T}')$ is connected at the $@^l$ m-node to the argument sub-cable of c and the result sub-cable of c is connected to the root of the graph.
- In the case for pairs, the cables from the roots of the components are combined at a $(\cdot,\cdot)^l$ value m-node. The cable leaving the $(\cdot,\cdot)^l$ has a pair type and goes to the root. The case for '$\text{let}^l\ (x,y)$ be e in e'' lets a pair cable from $\mathcal{G}(\mathcal{T})$ enter a '$\text{let}^l\ (\cdot,\cdot)$' consumer m-node and sends the sub-cables on to the (unique) m-nodes for the bound variables. The graph $\mathcal{G}(\mathcal{T}')$ connects directly to the root (since e' will usually have occurrences of x or y, the graph will usually be connected). The dotted edge from the '$\text{let}^l\ (\cdot,\cdot)$' m-node to $\mathcal{G}(\mathcal{T}')$ is only included to aid readability and is not part of the graph.
- The cases for booleans and 'if' should be straightforward — note that both branches of conditionals connect to the root, indicating that we do not know which branch will be taken.
- Applying $\mathcal{G}$ to $\frac{\mathcal{T}}{A \vdash \text{fix}^l x.e : t}$ connects $\mathcal{G}(\mathcal{T})$ to x indicating that the result of evaluating e is bound to x. We then connect x to the root. Note, that the 'fix' m-node is not connected to the graph — thus variable, data, consumer and box m-nodes suffice (the 'fix' m-node is only included for readability).

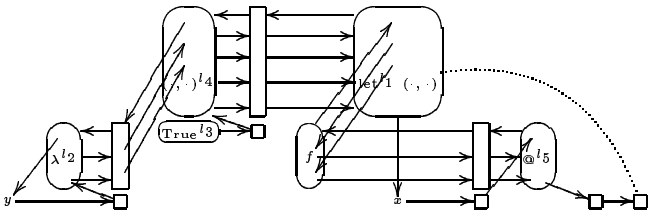
The *interface* of a graph $G = \mathcal{G}(\frac{\mathcal{T}}{A \vdash e : t})$ consists of the root m-node of G and the variable m-nodes x where x is free in e. Note that there is a one-to-one correspondence between type constructors in $A \vdash e : t$ and nodes in the interface of G. An occurrence of a type constructor in $x_1 : t_1, \cdots, x_n : t_n \vdash e : t$ is called positive (negative) if it occurs positively (negatively) in $t_1 \to \cdots \to t_n \to t$.[2] A node occurs positively (negatively) in the interface of $\mathcal{G}(\frac{\mathcal{T}}{A \vdash e : t})$ if the corresponding type constructor occurs positively (negatively) in $A \vdash e : t$.

Example 1. Applying $\mathcal{G}$ to the unique derivation of

$$\vdash \text{let}^{l_1}\ (f,x)\ \text{be}\ (\lambda^{l_2} y.y, \text{True}^{l_3})^{l_4}\ \text{in}\ f@^{l_5} x : \text{Bool}$$

results in the following typed flow graph:

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



The reader is encouraged to follow the def-use path from λ^{l_2} to $@^{l_5}$ and the path from the True^{l_3} to the root of the graph.

3.2 Polymorphism

To add polymorphism, we extend the language of types as follows[3]:

$$t ::= \tau \mid \mathrm{Bool} \mid t \to t' \mid t \times t' \mid \forall \tau.t$$

where we use τ for type variables. We add explicit syntax for generalisation and instantiation. The extension to the language is defined by the type rules

$$\forall\text{-I}\ \frac{A \vdash e : t}{A \vdash \Lambda^l \tau.e : \forall \tau.t}\quad (\tau \text{ not free in } A) \qquad \forall\text{-E}\ \frac{A \vdash e : \forall \tau.t'}{A \vdash e\{t\}^l : t'[t/\tau]}$$

and the reduction rule:

$$(\beta)\ (\Lambda^l \tau.e)\{t\}^{l'} \longrightarrow e$$

where Λ is data and $\{\}$ is a consumer.

Intuitively, a type variable τ can carry any value since it might be instantiated to any type. For graphs, however, the opposite intuition is more fruitful: *no* value is carried along a τ-cable, since, as long as the value has type τ, it *cannot* be used. This approach relies on the same intuition as "Theorems for Free", that a function cannot touch arguments of polymorphic type [12]. Thus a τ-cable is no cable at all and the appropriate connections are made at the instantiation m-node. A $\forall \tau.t$ cable is a t-cable.

Since t-cables and $\forall \tau.t$-cables are the same, a quantification m-node just passes on its incoming cable:

$$\mathcal{G}\left(\frac{\mathcal{T}}{A \vdash \Lambda^l \tau.e : \forall \tau.t}\right) \quad = \quad \mathcal{G}(\mathcal{T}) \Longrightarrow \boxed{\Lambda^l} \Longrightarrow \square$$

An instantiation m-node has an ingoing $\forall \tau.t'$ cable and an outgoing $t'[t/\tau]$ cable. All wires of the $\forall \tau.t'$ cable are connected to the similar wires in the $t'[t/\tau]$ cable. The remaining edges of the $t'[t/\tau]$ cable form t cables — these

[3] Since we are analysing programs that are already typed, System F gives a smoother presentation — the program might well be typed without using the full power (e.g. by allowing polymorphism only in let-bound expressions).

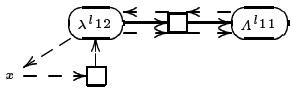
are connected such that negative occurrences of t are connected to all positive occurrences of t

To be precise, assume that t' has n occurrences of τ and write the occurrences as $\tau^{(1)}, \cdots \tau^{(n)}$ and the similar occurrences of t in $t'[t/\tau]$ as $t^{(1)}, \cdots t^{(n)}$. For any pair $\tau^{(i)}$, $\tau^{(j)}$ where $\tau^{(i)}$ occurs negatively and $\tau^{(j)}$ occurs positively in t', add a t-cable from $t^{(i)}$ to $t^{(j)}$.

Example 2. Consider

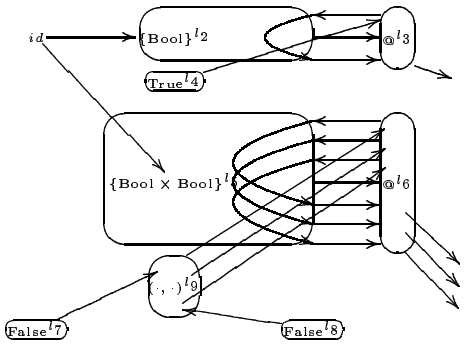
$$\lambda^{l_1} id.(id\{\text{Bool}\}^{l_2} @^{l_3} \text{True}^{l_4}, id\{\text{Bool} \times \text{Bool}\}^{l_5} @^{l_6} (\text{False}^{l_7}, \text{False}^{l_8})^{l_9})^{l_{10}} \\ @\Lambda^{l_{11}} \tau.\lambda^{l_{12}} x.x$$

where we assume that id is given type $\forall\tau.\tau \to \tau$. The graph fragment for the argument $\Lambda^{l_{11}} \tau.\lambda^{l_{12}} x.x$ looks as follows



The dashed edges are not part of the graph and are only included to make the graph more readable since it would otherwise be completely unconnected[4].

The graph fragment for the applications of id looks as follows:



(we have left out superfluous box-nodes).

3.3 Sum Types

Sum types have the following syntax:

$$t ::= \text{Bool} \mid t \to t' \mid t \times t' \mid t + t' \mid 1$$

Again, we associate syntax with the type rules:

$$1\ \frac{}{A \vdash u^l : 1} \qquad +\text{-I}\ \frac{A \vdash e : t}{A \vdash \text{inl}^l(e) : t + t'} \qquad \frac{A \vdash e : t'}{A \vdash \text{inr}^l(e) : t + t'}$$

$$+\text{-E}\ \frac{A \vdash e : t + t' \quad A, x : t \vdash e' : t'' \quad A, y : t' \vdash e'' : t''}{A \vdash \text{case}^l\ e\ \text{of inl}(x)\ \mapsto\ e';\ \text{inr}(y)\ \mapsto\ e'' : t''}$$

[4] While the dashed edges are not necessary to find def-use paths, they can be included if we want information about which values a given variable can be bound to.

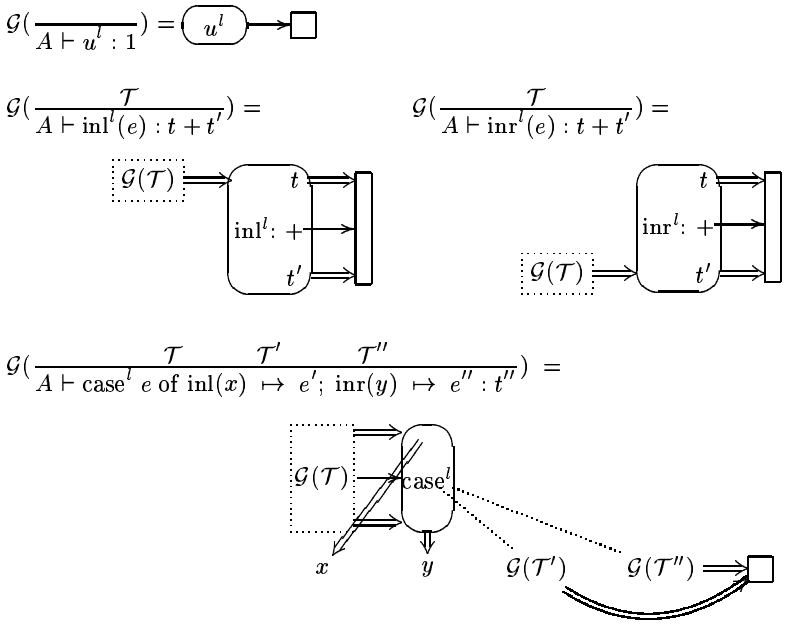


Fig. 4. Typed flow graphs for sum types

The reduction rules for the constructs are as follows:

$$(\delta\text{-case})\ \ \mathrm{case}^{l'}\ \mathrm{inl}^l(e)\ \mathrm{of}\ \mathrm{inl}(x) \mapsto e';\ \mathrm{inr}(y) \mapsto e'' \longrightarrow e'[e/x]$$
$$\mathrm{case}^{l'}\ \mathrm{inr}^l(e)\ \mathrm{of}\ \mathrm{inl}(x) \mapsto e';\ \mathrm{inr}(y) \mapsto e'' \longrightarrow e''[e/y]$$

where 'inl' and 'inr' construct data and 'case' is a consumer.

To extend typed graphs with sum types, we first have to define cables carrying values of sum type:

A $(t+t')$-cable is $\overset{1}{\underset{2}{\Rightarrow}}$ where $\overset{1}{\Longrightarrow}$ is a t-cable and $\overset{2}{\Longrightarrow}$ is a t'-cable.

The carrier cable represents the flow of the 'inl'/'inr' data.

In figure 4 we extend the definition of $\mathcal{G}$ with the new syntactic constructs. The constructs should be straightforward: the unit u^l is treated like other constants, $\mathrm{inl}^l(e)$ and $\mathrm{inr}^l(e)$ connect the root of $\mathcal{G}(\mathcal{T})$ to the appropriate sub-cable of the sum-cable — nothing flows into the other sub-cable. Finally, the case-construct decomposes the sum (in a manner similar to '$\mathrm{let}^l\ (\cdot,\cdot)$') and connects the branches to the root (similarly to 'if').

3.4 Recursive Types

Recursive types add the ability to define integers, lists etc.:

$$t ::= \tau \mid \mathrm{Bool} \mid t \to t' \mid t \times t' \mid t + t' \mid \mu\tau.t$$

where we have retained the sum types from above (since they are required to make practical use of recursive types).

Usually, recursive types are added to type systems by adding the equivalence $\mu\tau.t = t[\mu\tau.t/\tau]$ We make applications of this equivalence explicit in the language by adding syntax with the following type rules:

$$\text{fold}\ \frac{A \vdash e : t[\mu\tau.t/\tau]}{A \vdash \text{fold}^l(e) : \mu\tau.t} \qquad \text{unfold}\ \frac{A \vdash e : \mu\tau.t}{A \vdash \text{unfold}^l(e) : t[\mu\tau.t/\tau]}$$

We consider 'fold' as data and 'unfold' as a consumer and add the reduction rule:

$$(\delta\text{-rec})\ \text{unfold}^{l'}(\text{fold}^l(e)) \longrightarrow e$$

As with polymorphism, τ cables are empty — this makes even more sense with recursive types as no variable will ever have type τ and hence the values that a variable can evaluate to can be read from the graph even without τ-cables.

A $\mu\tau.t$ cable has to carry the information from all unfoldings of the type, hence we need a t cable to carry the information of t as well as instantiations with $\mu\tau.t$ of positive occurrences of τ, and a flipped t cable to carry the information of instantiations of negative occurrences of τ. Similarly, we need a wire in each direction carrying 'fold' values. Thus

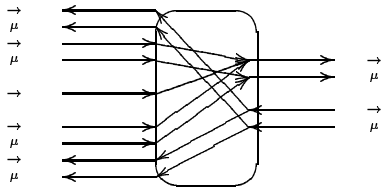
A $\mu\tau.t$ cable is  where  is a t-cable,  is its flipped version and  is a t-cable.

The forward single edge is the carrier and carries the label of the applied fold operation; the backward single edge carries the labels of all fold operation that can occur in argument position.

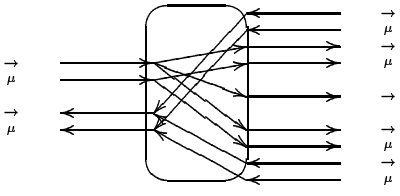
Fold and unfold m-nodes are dual and parameterised by the recursive type involved. An unfold m-node has an incoming $\mu\tau.t$ cable and an outgoing $t[\mu\tau.t/\tau]$ cable. Let superscripts index the occurrences of τ in t as in the polymorphic case. We connect the edges of the positive sub cable of the incoming cable to the nodes of the outermost t on the outgoing side. Furthermore, the incoming $\mu\tau.t$ cable is connected to all $\mu\tau.t^{(i)}$ — directly if $\tau^{(i)}$ is a positive occurrence of τ in t and "switched" if $\tau^{(i)}$ is a negative occurrence. The fold m-node has an incoming $t[\mu\tau.t/\tau]$ cable and an outgoing $\mu\tau.t$ cable. Connections are made similarly.

Example 3. The 'fold' m-node for folding $(\mu\tau.\tau \to \tau) \to (\mu\tau.\tau \to \tau)$ to $\mu\tau.\tau \to \tau$ and the dual 'unfold' m-node for unfolding $\mu\tau.\tau \to \tau$ to $(\mu\tau.\tau \to \tau) \to (\mu\tau.\tau \to \tau)$ are given below. The type constructors are included to remind the reader of the kind of labels carried by the individual wires.

fold:



unfold:



$$Bool! : \mathrm{Bool} \rightsquigarrow \mathsf{D} \qquad Fun! : \mathsf{D} \to \mathsf{D} \rightsquigarrow \mathsf{D} \qquad Pair! : \mathsf{D} \times \mathsf{D} \rightsquigarrow \mathsf{D}$$
$$Bool? : \mathsf{D} \rightsquigarrow \mathrm{Bool} \qquad Fun? : \mathsf{D} \rightsquigarrow \mathsf{D} \to \mathsf{D} \qquad Pair? : \mathsf{D} \rightsquigarrow \mathsf{D} \times \mathsf{D}$$

$$\frac{c_1 : t_1 \rightsquigarrow t_1' \quad c_2 : t_2 \rightsquigarrow t_2'}{c_1 \times c_2 : t_1 \times t_2 \rightsquigarrow t_1' \times t_2'} \qquad \frac{c_1 : t_1 \rightsquigarrow t_1' \quad c_2 : t_2 \rightsquigarrow t_2'}{c_1 \to c_2 : t_1' \to t_2 \rightsquigarrow t_1 \to t_2'} \qquad \mathrm{Sub}\ \frac{A \vdash e : t \quad c : t \rightsquigarrow t'}{A \vdash [c]^l e : t'}$$

Fig. 5. Dynamic Typing

3.5 Dynamic Types

For dynamic typing we need simple types plus the special type D:

$$t ::= \mathsf{D} \mid \mathrm{Bool} \mid t \to t' \mid t \times t'$$

The necessary extensions to the type system of figure 1 are given in figure 5 (for more details see Henglein [5]).

The conversions $Bool!, Fun!, Pair!$ correspond to tagging operations: they take an untagged value which the type system guarantees has a given type (eg. Bool) and throws it into the common pool of values about which the type system knows nothing. In this pool values have tags that can be checked at run time. Conversions $Bool?, Fun?, Pair?$ check the tag of a value and provide the untagged value of which the type inference now knows the type.

Using recursive types and sum types, we already have sufficient power to specify dynamic types. Type D is equivalent to

$$\mu\tau.((\tau \to \tau) + ((\tau \times \tau) + \mathrm{Bool}))$$

The conversions are expressible in our language. E.g. $[Bool!]^l$ is $\mathrm{fold}^l \circ \mathrm{inr}^l \circ \mathrm{inr}^l$ and $[Bool?]^l$ is $\mathrm{outr}^l \circ \mathrm{outr}^l \circ \mathrm{unfold}^l$ where outr^l is a shorthand for

$$\lambda^l x.\mathrm{case}^l\ x\ \mathrm{of}\ \mathrm{inl}(y)\ \mapsto\ error;\ \mathrm{inr}(z)\ \mapsto\ z$$

(having different labels on the sum and recursive type operators would not give any additional information, so we just assume that they are the same).

By the coding of dynamic types using sums and recursive types, we find that a D-cable consists of 6 forward and 6 backward edges (the 6 are $\to$, $+$, $\times$, $+$, Bool, μ). Since the labels carried by the sum and μ edges are the same, we can replace them with one forward and one backward edge carrying tag-labels:

where the edges read from top to bottom carry labels of type: !, $\to$, $\times$, Bool, !, $\to$, $\times$ and Bool. Now the tagging and untagging m-nodes can be found by combining m-nodes for sum and recursive types.

Example 4. The $[Fun!]^l$ m-node (equivalent to $\mathrm{fold}^l \circ \mathrm{inl}^l$) is given in figure 6 (where the left-hand side is a $\mathsf{D} \to \mathsf{D}$ cable and the right-hand side is a D cable).

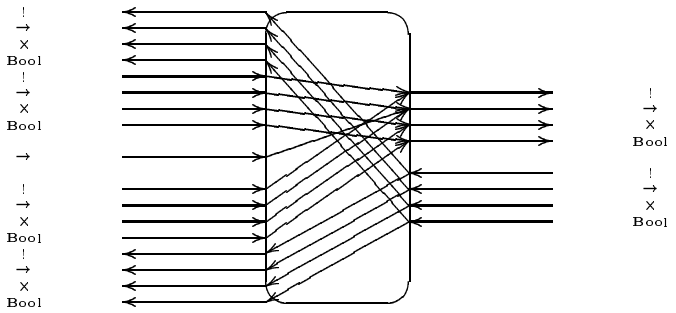


Fig. 6. The $[Fun!]^l$ m-node

3.6 Soundness

A flow graph is a sound abstraction of the definitions and uses of data during evaluation of an expression e, if (a) whenever there is a redex in e, there is a path starting from the data being consumed and ending at the consumer and (b) reduction does not add extra paths. We state soundness for the system with polymorphism, sums and recursive types. Soundness for the dynamic system follows by the construction. The proof of the following theorem can be found in [7].

Theorem 5. *Let C,e_1,e_2,$\mathcal{T}_1$,A,t be given such that $\frac{\mathcal{T}_1}{A \vdash C[e_1] : t}$ and e_1 reduces to e_2 by a β or δ redex. Then there exists $\mathcal{T}_2$ such that*

1. $\frac{\mathcal{T}_2}{A \vdash C[e_2] : t}$
2. *if the redex lets l consume l' then there is a def-use path from m-node l' to m-node l in $\mathcal{G}(\frac{\mathcal{T}_1}{A \vdash C[e_1] : t})$ and*
3. $\{(l_1, l_2) \mid$ *there is a def-use path from l_1 to l_2 in $\mathcal{G}(\frac{\mathcal{T}_1}{A \vdash C[e_1] : t})$*$\}$
   $\supseteq \{(l_1, l_2) \mid$ *there is a def-use path from l_1 to l_2 in $\mathcal{G}(\frac{\mathcal{T}_2}{A \vdash C[e_2] : t})$*$\}$

4 Discussion

4.1 Algorithm

So far we have only discussed the construction of flow graphs. It should be clear, however, that the usual flow problems ("Where is a given occurrence of a value used?", "Which values are used by a given consumer?" and the full flow problem) can be solved by standard reachability algorithms. Single data or single consumer problems can be answered in linear time while the full flow problem can be answered in quadratic time. In the same way, we can answer questions

such as "which values can a given expression evaluate to?", or "which values can a given variable be bound to?", though some care should be taken for the latter query in the polymorphic case (as mentioned in section 3.2).

We can leave out wires for polymorphism, sums, recursiveness etc. if tracing this is not necessary. Further, if only what is usually known as control flow information[5] is needed then more can be left out. A $\mu\tau.t$-cable does not need the backward t-cable and μ-edge if there are no negative occurrences of τ in t.

It is also possible to reduce the size of graphs by eliminating unnecessary nodes: most box-nodes and the quantification nodes can immediately be removed, but if we are only interested in part of the flow (e.g. only higher-order flow) more can be removed. By reachability (see Ayers [2]) we can remove cables in 'if' and 'case' provided the flow graph can prove that all values destructed will be True or all values will be False (resp. inl$(\cdot)$ or inr$(\cdot)$). Furthermore, if we are only interested in the input/output behaviour, the graph can be reduced to depend only on the size of the interface (i.e. constant under assumption that types are bounded). This is useful if further polyvariance is desired as cloning the graphs for a definition will be cheap (this option is discussed by Heintze and McAllester [4]).

4.2 Precision

It was shown in [6] that the simply typed fragment of this analysis (that is the analysis presented in section 3.1) was equivalent to closure analysis of the same program. The equivalence extends in a straightforward manner to sum types.

Assume that e is polymorphically typed. Our analysis can yield better results than closure analysis when analysing e: using the polymorphic type information gives us a degree of polyvariance. Consider the identity function $id = \lambda^{l_1} x.x$ of type $\forall\tau.\tau \to \tau$ and assume two call sites $id@^{l_2}\text{True}^{l_3}$ and $id@^{l_4}\text{False}^{l_5}$. Closure analysis will find that the result of both applications can be l_3 or l_5. Our analysis will create two instantiation nodes and hence keep the applications separate: the result will be l_3 for the first application and l_5 for the second. If the identity function was given type Bool $\to$ Bool instead, we would get the same (sub-optimal) result as closure analysis. This loss of polyvariance would also arise if the identity was specified as $\lambda^{l_1} x.\text{if}^{l_2}\ x$ then x else x since this can only be given a monomorphic type.

Consider recursive types. Our analysis is *potentially* as good as closure analysis. Heintze shows that flow analysis based on recursive types is equivalent to closure analysis [3]. It is not difficult to show that our analysis is equivalent to recursive type based analysis, *except* that the base types are chosen in advance. This implies that for every untyped term e, there *exists* a type derivation for e such that the resulting graph contains information equivalent to closure analysis.

Example 6. Consider lists. The usual type for lists of element of type t is $\mu\tau.((t \times \tau) + 1)$. Using this type, $[\text{True}^{l_1}, \text{False}^{l_2}]$ is shorthand for

$$\text{fold}(\text{inl}((\text{True}^{l_1}, \text{fold}(\text{inl}((\text{False}^{l_2}, \text{fold}(\text{inr}(u^{l_0})))^{l_6})))^{l_7}))$$

[5] We prefer to think of this as value flow of control structures

(where we have left out some unnecessary labels). Using this type for lists, our analysis will find that $\mathrm{fst}([\mathrm{True}^{l_1}, \mathrm{False}^{l_2}])$ (where fst is shorthand for $\lambda x.\mathrm{let}\ (f,s)$ be $(\mathrm{case}\ \mathrm{unfold}(x)\ \mathrm{of}\ \mathrm{inl}(y) \mapsto y;\ \mathrm{inr}(z) \mapsto \mathit{error})$ in f) can result in l_1 as well as l_2. Using a different type for lists (such as $(t \times \mu\tau.((t \times \tau)+1))+1$) will yield the precise result in this case.

4.3 Related Work

Independently of this work, Heintze and McAllester developed a flow analysis which has the same complexity as ours on simply typed programs [4] (on untyped programs it might not terminate). Their analysis is derived by transforming the constraint formulation of flow analysis [8] to avoid computation of dynamic transitive closure. For each node n there are potentially nodes $\mathit{dom}(n)$ and $\mathit{ran}(n)$. Due to the boundedness of standard types, the number of nodes is similarly bounded — this corresponds directly to our m-nodes.

Types, however, are treated fundamentally different in their presentation. While types is an integrated part of the generation of our flow graphs, it only appears in the proof of termination/complexity of their algorithm. We believe that this difference make it difficult to extend their system. In particular:

Polymorphism Since their algorithm does not need the types, it is directly applicable to polymorphically typed programs. Complexity results, however, are preserved only if the monotypes of the expanded program is bounded by a constant. While this is not unreasonable for let-polymorphic languages, our solution preserves the complexity results if the size of types of the *original* program is bounded by a constant. Furthermore, we get polyvariance for free.

Recursive types Heintze and McAllester shows how their analysis can be extended to handle lists. It is not clear, however, how to generalise to arbitrary recursive types — for types $\mu\tau.t$ where τ occurs negatively in t the backward cable is a necessary and non-obvious extension which does not fit easily into their framework. As a consequence their analysis is not applicable to dynamically typed programs (as mentioned above, termination is not guaranteed if the analysis is applied to untyped terms).

We find that our improvements rely heavily on our formalism and in particular on the availability of types at graph construction time.

The notion of *well-balanced path* by Asperti and Laneve [1] corresponds directly to the usual constraint formulation of flow analysis. Asperti and Laneve refine the concept of well-balanced path to *legal path* which captures exactly the set of virtual redexes in a term — a notion of *exact* flow analysis useful for optimal reduction. If we place a restriction similar to legality on the paths of our graphs, we will obtain an exact flow analysis — as a direct consequence of the precision, the analysis will be non-elementary recursive. We hope to find a formulation of the legality restriction, that is amenable to abstraction and hence develop analyses strictly more precise than closure analysis, but with a

non-prohibitive complexity. In particular, we hope to find the first-order analysis of Horowitz, Reps and Sagiv [9] as a special case.

5 Conclusion

We have presented a notion of flow graph for higher-order programs. Under assumption that the size of all types is bounded by a constant, the resulting flow analysis presents an asymptotic improvement over earlier work. Heintze and McAllester have independently of this work reported similar results, but in contrast to their work, our approach handles recursive types and is hence applicable to untyped programs (if a dynamic type discipline is enforced). Furthermore, we handle polymorphism in a more general way which entails a desirable degree of polyvariance.

Acknowledgements This paper extends work presented in my Ph.D.-thesis [6]. I would like to thank my supervisor Fritz Henglein for many rewarding discussions and my official opponents, Alex Aiken, Nils Andersen and Peter Sestoft for their comments and suggestions.

References

1. A. Asperti and C. Laneve. Paths, computations and labels in the λ-calculus. In *RTA '93*, volume 690 of *LNCS*, pages 152–167. Springer-Verlag, 1993.
2. A. Ayers. Efficient closure analysis with reachability. In *Proc. Workshop on Static Analysis (WSA), Bordeaux, France*, pages 126–134, Sept. 1992.
3. N. Heintze. Control-flow analysis and type systems. In A. Mycroft, editor, *Symposium on Static Analysis (SAS)*, volume 983 of *LNCS*, pages 189–206, Glasgow, 1995.
4. N. Heintze and D. McAllester. Control-flow analysis for ML in linear time. In *International Conference on Functional Programming (ICFP)*, 1997. To appear.
5. F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)*, 22(3):197–230, 1994.
6. C. Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, University of Copenhagen, January 1997.
7. C. Mossin. Higher-order value flow graphs. Technical report, DIKU, 1997.
8. J. Palsberg. Global program analysis in constraint form. In S. Tison, editor, *19th International Colloquium on Trees in Algebra and Programming (CAAP)*, volume 787 of *LNCS*, pages 276–290. Springer-Verlag, 1994.
9. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, CA, Jan. 1995.
10. P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.
11. P. Sestoft. *Analysis and Efficient Implementation of Functional Languages.* PhD thesis, DIKU, University of Copenhagen, Oct. 1991.
12. P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, Sept. 1989.