A New Approach to Control Flow Analysis

Pasquale Malacaria and Chris Hankin

Dept. of Computing
Imperial College
LONDON SW7 2BZ
e-mail: pm5,clh@doc.ic.ac.uk

Abstract. We develop a control flow analysis algorithm for PCF based on game semantics. The analysis is closely related to Shivers' 0-CFA analysis and the algorithm is shown to be cubic. The game semantics basis for the algorithm means that it can be naturally extended to handle strict languages and languages with imperative features. These extensions are discussed in the paper. We sketch the correctness proof for the algorithm. We also illustrate an algorithm for computing k-limited CFA.

1 Introduction

Many optimisations are based on some representation of the control flow in a program. This is explicitly recognised in the standard presentations of classical data flow analyses [10, 3, 11, 21]. For higher-order languages, or languages with concurrency primitives, it is not easy to determine the control flow: control flow analysis (CFA) aims to determine which closures appear at which points in the execution of the program. Examples of the use of CFA include:

- effect analysis [14]: that finds which expressions in ML programs have side effects.
- k-limited CFA [14]: if there are up to k functions that may be called at a particular point, they are listed; otherwise the analysis indicates that "many" functions may be called (no explicit listing).
- called-once analysis [14]: that identifies functions which are only called from one point.
- uncaught exceptions analysis [29]: that analyses ML programs to identify exceptions that may be raised but not caught.
- locality analysis [9]: that analyses programs written in a concurrent object calculus to determine if method invocations are local or remote.

The problem of determining control flow in first-order languages is almost trivial. It is a much more difficult problem for higher-order languages where functions may be passed as parameters and invoked from anywhere in the program. Over recent years there has been intense activity in designing CFAs for higher-order languages [4, 5, 12, 16–18, 20, 22, 23, 26, 27]. There has been convergence on the definition of the fundamental notion of CFA which has been dubbed *standard CFA* by [14].

A number of algorithms have been proposed for standard CFA. The "best" algorithms are cubic; recent work by Heintze and McAllester suggests that this situation cannot be improved [13]. However, as with ML type-checking, this worst-case behaviour is seldom observed in practice. In another paper [14], Heintze and McAllester present a linear-time algorithm for simply typed λ-calculus that uses a graph structure and which may be used to give linear-time algorithms for many of the applications listed in the first paragraph. When extended to a simply typed λ-calculus with conditionals and constants, their algorithm is no better than standard CFA. Independently, we have developed a CFA algorithm that also uses a graph structure. Furthermore, our algorithm is derived from a simple abstraction of the game semantics of the program under analysis – correctness is therefore "by construction". The main body of the paper presents the analysis for PCF (a typed, higher-order, call-by-name functional language), but strict languages and languages with imperative and concurrent features can be accomodated within the same game-theoretic framework.

Game Semantics came to prominence in 1994 when they were used to solve a long-standing open problem in the field of programming language semantics (the full abstraction problem for PCF) [1]. Games inhabit a category which has a rich mathematical structure; a consequence is that Games Semantics has an extensive proof theory, a feature shared with denotational semantics. On the other hand, Games Semantics contains a lot of intensional information about a program, a feature shared with operational semantics. The combination of these two properties makes Game Semantics an ideal candidate as a basis for semantics-based program analysis.

We consider two player games; the Player is the program and the Opponent is the environment. Moves may be questions – requests for results (Opponent questions) or inputs (Player questions) – or answers. A game may be characterised by a set of labelled moves and a set of valid positions (sequences of moves). Every play of the game starts with an Opponent question. A strategy for Player is a set of even length positions which satisfies some additional requirements (depending on the language being modelled). The strategies used in our work can be conveniently represented using trees.

The algorithm has several parts, the main ones being:

1. A graph is built for each normal form; this graph is an "abstract strategy" [1].
2. The graphs are linked; these links corresponds to composition in the category of games and are related to the linear head reduction of the main term [7].
3. An environment and cache [22] are constructed from the resulting graph.

The approaches to CFA in the literature can be categorised under four headings:

Abstract Machines: Historically this was one of the first approaches [18]. Closure information is collected by abstracting the states of an abstract machine – an SECD-like machine in the case of Jones [18] and the Geometry of Interaction Machine in the case of Jensen and Mackie [17].

Abstract Semantics: In this approach, closure information is collected by a non-standard semantics, which is sometimes presented as an abstract interpretation of the standard semantics. Two early examples are [5, 26] which use non-standard denotational semantics. Shivers [27] and Jagannathan and Weeks [16] abstract from an operational semantics.

Constraint-based: A number of authors have used intensional semantics (usually operational semantics) to generate constraints that describe control flow. Heintze [12], Nielson and Nielson [22] and Palsberg [23] are examples of this approach.

Type-based: Recently, a number of authors have proposed the use of annotated types to capture control flow information. Mossin [20] and Banerjee [4] are good examples of this approach.

Our approach is essentially based on abstract semantics and is related also with the abstract machines approach. These relationships arise from the particular nature of the game model where types are interpreted as some games in a (cartesian closed) category whose strategies are the interpretation of programs (e.g. the interpretation of `succ` is the strategy in the game `int` $\rightarrow$ `int` where Player plays the move $n+1$ whenever Opponent plays n). This intensional character of game semantics (programs are interpreted by strategies and no more by functions, although function can then be recovered by collapsing strategies according to logical relations) has played a key role in the solution of the "full abstraction problem" for PCF and relies on a technical lemma (the decomposition lemma) which is also the key tool in the analysis here presented.

The rest of this paper is structured as follows. In the next section we present the steps of the algorithm and consider its complexity. In Section 3, we consider the correctness of the algorithm. We sketch the correctness based on game semantics. In Section 5, we consider two extensions to the basic algorithm:

1. We describe a transformation which provides a safe estimation of control flow for strict languages.
2. We describe the extension for CFA in the presence of imperative features using *Idealised Algol* [25] as an exemplar of the techniques.

We then show how one of the analyses described in the first paragraph of this section can be formalised in our framework. We conclude with a review of the achievements of the analysis presented in this paper and directions for further work.

2 The analyser

2.1 The language PCF

PCF [19] is a simply typed lambda calculus with arithmetic and boolean constants; terms and types are defined as follows:

– Types are: $T ::= B|T \rightarrow T$ where B is a basic type `int` or `bool`.

- Terms are:
  - For each type there is an infinity of variables of that type (noted as $x^T, y^T, \ldots$)
  - If $M^{T \to T'}, N^T$ are terms, $\lambda x^U.N$ is a term of type $U \to T$ and MN is a term of type T'.
  - There are basic type constants $0, 1, \ldots$ of type `int` for each natural number and `tt`, `ff` of type `bool`.
  - There are first order constants `succ`, `pred` of type `int` $\to$ `int` for the successor and predecessor functions and a test for zero `iszero` of type `int` $\to$ `bool`. There is also a constant for conditional expressions `cond` of type `bool` $\to B \to B \to B$.
  - There is a family of fixpoint constants $\mathbf{Y_T}$, one for each type $T = A_1 \to \ldots \to A_n \to B$. [1]

The notion of reduction we use is *Linear Head Reduction*[7]. Intuitively the Linear Head Reduction of a term M can be described as follows:

- At each step consider the leftmost head variable x in the term M[2].
- Replace (only) this occurrence of x with a copy of the corresponding term N (correspondence given by λ binding).
- If this was the only occurrence of x then erase the original N and the corresponding λx.

For example (where $\to_{\mathtt{lhr}}$ denotes one step of linear head reduction):

$$\begin{aligned}(\lambda x.x(x\mathtt{tt}))(\lambda y.y) &\to_{\mathtt{lhr}} (\lambda x.(\lambda y.y)(x\mathtt{tt}))(\lambda y.y)\\ &\to_{\mathtt{lhr}} (\lambda x.x\mathtt{tt})(\lambda y.y)\\ &\to_{\mathtt{lhr}} (\lambda y.y)\mathtt{tt}\\ &\to_{\mathtt{lhr}} \mathtt{tt}\end{aligned}$$

2.2 Representing terms

For sake of simplicity we use a (linear) transformation of PCF programs so that every program has the form[3].

$$(*) \quad M_1 \ldots M_n$$

[1] For our purposes we define $Y_T = \lambda f x_1 \ldots x_n.\mu_T(f)x_1 \ldots x_n$ where μ_T is a new constant whose operational rule is $\mu_T(f)$ evaluates to $f(\mu_T(f))$ for a variable f of type $T \to T$. This rule makes sense because we are using Linear Head Reduction which allows us to substitute one occurrence of f at the time.

[2] The generalisation to delta rules is trivial (think of the head *component* instead the head variable).

[3] A trivial method to transform an arbitrary closed term into one of the shape $(*)$ is to use the *normalised form*. Given a term M, the normalised form $\bar{M}$ is defined as follows:

- First define $\bar{M}^0 = M[r_i R_i C_i / R_i C_i]$ where $R_i C_i$ is the $i-$th redex in M and r_i is a fresh variable ($1 \leq i \leq n$ with n being the number of redexes in M).
- Define then $\bar{M} = (\lambda r_1 \ldots r_n.\bar{M}^0)I_1 \ldots I_n$ where I_i is the identity of type $A_i \to A_i$ (with A_i being the type of R_i).

where each M_i is in normal form and the whole term is closed.

Given a normal form M we define its inductive translation $[M]$ as follows:

- Normal forms of basic type are translated by the empty graph.
- $M = \lambda x_1 \dots x_m . y M_1 \dots M_n$

$$[M] = \begin{tikzcd} & M \arrow[d] & \\ & y \arrow[dl] \arrow[dr] & \\ {[M_1]} & \cdots & {[M_n]} \end{tikzcd}$$

- $M = \lambda x_1 \dots x_m . \texttt{cond}\ (y M_1 \dots M_n)\ N\ N'$

$$[M] = \begin{tikzcd} & & M \arrow[d] & & \\ & & y \arrow[dll] \arrow[d] \arrow[dr] \arrow[drr] & & \\ {[M_1]} & \cdots & {[M_n]} & {[N]} & {[N']} \end{tikzcd}$$

- $M = \mathbf{Y} : (A \to A) \to A$ where $A = A_1 \to \dots \to A_n \to B$ with B a basic type. Then by definition $\mathbf{Y} = \lambda f x_1 \dots x_n . \mu(f) x_1 \dots x_n$ and

$$[\mathbf{Y}] = \begin{tikzcd} & & & \mathbf{Y} \arrow[d] \\ & & & q \arrow[dlll] \arrow[dl] \arrow[d] \\ {[x_1]} & \cdots & {[x_n]} & q' \arrow[u, bend right] \end{tikzcd}$$

The other delta rules $\texttt{succ}, \texttt{pred}, \texttt{iszero}$ are all interpreted by the same tree which is:

$$\begin{tikzcd} \mathbf{T} \arrow[d] \\ q \end{tikzcd}$$

We will call *variables* the vertices at an even level of the tree and *subterms* the vertices at an odd level of the tree (the root of a tree is at level 1, for the fixpoint graph apply the convention of erasing the looping (i.e. the upward) edge). Notice that in the above translations we have introduced new variables q and subterms q', they represent moves from the corresponding strategies which are not derived from the syntax – their rôle will become clear later.

For example if $M = (\lambda x.x(x0))(\lambda y.1)$ then $(\lambda x.x(x0))$ has type $(N \to N) \to N$ so $\bar{M} = (\lambda r.r(\lambda x.x(x0))(\lambda y.1))(\lambda fv.fv)$ with $(\lambda fv.fv)$ being the identity of type $((N \to N) \to N) \to (N \to N) \to N$.

2.3 Interaction links

The graphs above defined correspond to "abstract strategies". We are now going to define on such structure the abstract linking between graphs given by the composition of strategies in the category of games (remember that composition in a CCC "corresponds" to substitution in a typed lambda calculus). Given strategies $\sigma : A \to B$ and $\tau : B \to C$ composition copies a Player (resp. Opponent) question in the subgame B of the game $B \to C$ as an Opponent (resp. Player) question in the subgame B of the game $A \to B$.

The way in which this copycat is interpreted in our framework is by connecting vertices of different graphs built in the previous section through new links. For $M_1 \dots M_n$ as above here is how these links are created.

- Notice first that each variable and subterm in $M_1 \dots M_n$ has associated a unique occurrence of a type. The difference between type and occurrence is essential for the accuracy of the analysis. For lambda terms and conditional expressions these types are given by the syntax of PCF; for the delta rules these occurrences are: The root $\mathbf{T}$ has type $\texttt{int} \to \texttt{int}$ for $\texttt{succ}, \texttt{pred}$ and $\texttt{int} \to \texttt{bool}$ for $\texttt{iszero}$; for the variable q, the type is $\texttt{int}$ (the leftmost occurrence) for all three. For the fixpoint $\mathbf{Y} : (A \to A) \to A$ the root has type $(A \to A) \to A$, the variable q has type $(A \to A)$ and the subterm q' has type A (the leftmost occurrence).
- Let $A_2 \to \dots \to A_r \to B$ be the type of M_1 so that M_j $(j > 1)$ has type A_j. Mark as twin these two occurrences of A_j.
- For all M_i, $1 \le i \le n$ do the following association of variable and subterms with occurrence of subtypes as follows:
  0 Variables are associated with the occurrence of their types.
  1 If x is associated with $A'_1 \to \dots \to A'_m \to B$ and $N_1 \dots N_h$ are arguments of x (i.e. $xN_1 \dots N_h$ is a subterm) then N_j is associated with A'_j.
  2 Moreover if $N_j = \lambda y_1 \dots y_k.M''$ and $A'_j = C_1 \to \dots \to C_h \to B$ Then y_l is associated with C_l.
  3 Repeat the same process for each N_j in $1 \le j \le h$ going back to the step 1.

Example: let $x(\lambda y.y\texttt{tt})$ be a term where x has type $((\texttt{bool} \to \texttt{bool}) \to \texttt{bool}) \to \texttt{bool}$ and y has type $\texttt{bool} \to \texttt{bool}$. Then the associations given above on the type of x are:

$$\overbrace{(\overbrace{(\overbrace{\overbrace{\texttt{bool}}^{\texttt{tt}} \to \texttt{bool}}^{y}) \to \texttt{bool}}^{\lambda y.y\texttt{tt}}) \to \texttt{bool}}^{x}$$

Given a variable x in M_1 (resp. a variable y in M_i, $i > 1$) which is associated with the occurrence of a subtype T of A_i, link x (resp. y) with the subterms in

M_i $(i > 1)$(resp. the subterms in M_1) which are associated with the occurrence of the same subtype.

An important remark is in order at this point. In order for twin types to have all the "matching pairs" some η expansions may be needed. Although these expansions can be minimised (we need to η expand only some variables of higher order type), for sake of simplicity we give a general definition:

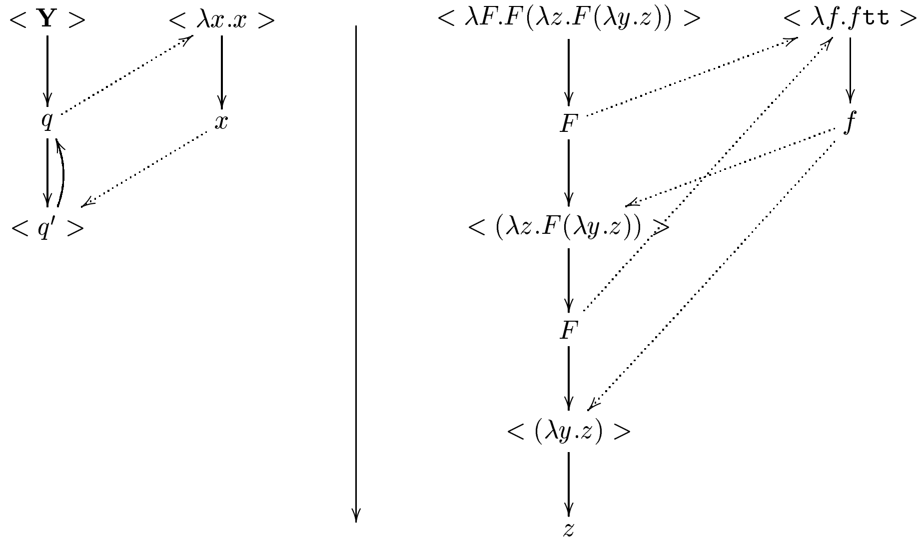
The η-expansion of a term is then defined in the following way:

$$\begin{array}{ll}
\eta(x^{\alpha_1 \dots \to \alpha_n \to B}) & = \lambda a_1 \dots a_n.x\ \eta(a_1) \dots \eta(a_n) \\
\eta(\lambda x.M) & = \lambda x.\eta(M) \\
\eta(M\ N) & = \eta(M)\ \eta(N) \\
\eta(\mu(f)\ M_1 \dots M_n) & = \mu(f)\ \eta(M_1) \dots \eta(M_n)
\end{array}$$

where B is a ground type (`int` or `bool`).

We will say that $z \in \texttt{d-arr}(x)$ if there is a link created by the previous procedure from x to z .

We conclude this section with two examples showing the machinery at work (dotted arrows represents interaction links). Here are the structures associated to $\mathbf{Y}_{\texttt{bool}\to\texttt{bool}}(\lambda x.x)$ (left) and $(\lambda F.F(\lambda x.F(\lambda y.x)))(\lambda f.f\texttt{tt})$ (right); we use $< t >$ to indicate the subterm vertices:



2.4 Tables

Given the above structure the algorithm computes the following recursively defined function on variable labeled vertices of the graph ($\mathcal{V}$ is the environment and $\mathcal{C}$ is the cache of [22]):

$$\mathcal{V}(x) = \bigcup_{z \in \texttt{d-arr}(x)} \begin{cases} \{\lambda x.M\} & \text{if } z = [\lambda x.M] \\ \bigcup(\mathcal{V}'((\mathcal{V}(y))^{\text{n-cut}})) & \text{if } z = [yM_1 \dots M_n] \\ \mathcal{V}'(N) \cup \mathcal{V}'(N') & \text{if } z = [\texttt{cond}MNN'] \end{cases}$$

where n-cut and $\mathcal{V}'$ are the obvious extension to sets of the following maps:

$$\mathcal{V}'(M) = \begin{cases} \{\lambda x.N\} & \text{if } M = [\lambda x.N] \\ \mathcal{V}'((\mathcal{V}(y))^{\text{n+r-cut}}) & \text{if } M = [(yM_1 \ldots M_n)]^{\text{r-cut}} \end{cases}$$

$$(\lambda x.M)^{\text{n+1-cut}} = M^{\text{n-cut}}, \quad M^{\text{0-cut}} = M$$

Concerning delta rules we only need to add the following equation for the fixpoint subterm vertex: $\mathcal{V}(q') = (\mathcal{V}(q))^{\text{1-cut}}$.

Note that computing $\mathcal{V}(x)$ it can happen to find again $\mathcal{V}(x)$; in this case we abort that branch of computation, i.e. we are taking the least solution.

The $\mathcal{V}$ map on variables is sufficient to define the cache map for arbritary program points. The cache map $\mathcal{C}$ is defined[4] as follows:

- $\mathcal{C}(\lambda x.M) = \{\lambda x.M\}$
- $\mathcal{C}(MM_1 \ldots M_r) = \mathcal{V}'(\{M\}^{\text{r-cut}})$
- $\mathcal{C}(x) = \mathcal{V}(x)$
- $\mathcal{C}(\texttt{cond}MNN') = \mathcal{C}(N) \cup \mathcal{C}(N')$

Applying the map $\mathcal{V}$ to the two previous examples we find the following values:

- $\mathcal{V}(x) = \emptyset$
- $\mathcal{V}(F) = \{\lambda f.f\texttt{tt}\}$
- $\mathcal{V}(f) = \{\lambda x.F(\lambda y.x), \lambda y.x\}$

2.5 Complexity of the algorithm

The analysis algorithm (i.e. the map $\mathcal{C}$) is cubic in n, the size of the term. This can be easily seen by considering that for each variable x, one element of $\mathcal{V}(x)$ can be computed in time n (since no vertex is visited more than once). Hence $\mathcal{V}(x)$ is computed in quadratic time and $\mathcal{C}$ in cubic time. The representation over which the analyser works is built in quadratic time. Note that these are the same bounds found in the literature for set contraints: building the contraints takes quadratic time and solving the contraints takes cubic time.

3 Correctness

Since the cache map $\mathcal{C}$ is defined in terms of the enviroment map $\mathcal{V}$ we state the result of this section in terms of the latter.

Proposition 1. *Let M be a PCF term and C be a closure substituted for the variable x in the Linear Head Reduction of M. Then there exists a closure $C' \in \mathcal{V}(x)$ and a substitution σ of the free variables of C' such that $C = \sigma(C')$.*

[4] For sake of completeness we stipulate that for a program point in normal form of basic type its cache is the empty set.

The proof of correctness of the algorithm relies heavily on Game semantics [1] and its relationship to Linear Head Reduction [7]. The starting point is how to relate our framework with game semantics: By using the decomposition lemma [1], we are going to translate our graphs as follow:

- Games are identified with occurrences of types.
- Variables of type $A_1 \to \ldots \to A_n \to B$ (with B a basic type) are identified with the unique Player question in the (rightmost occurrence of) the subgame B; subterms are identified with Opponent questions according to the same law.

Soundness is then a consequence of the following equation

$$R(F(\sigma^\circ \| \tau^\circ)) = F(\sigma^\circ) \|^\circ F(\tau^\circ)$$

which can be interpreted as follow:

- given a strategy σ, σ° is its approximation where "all answers have been removed".
- $\|$ is the parallel composition of strategies [1], whereas $\|^\circ$ is the `d-arr` relation described in section 2.4.
- F is the extension to sequences of moves of the map that erase all exponential indices[5] associated to a move.
- R is the relation between moves induced by the copycat of the parallel composition.

The above equation should hence be read as follows:

"Take the set of positions in $(\sigma^\circ \| \tau^\circ)$ and forget the indices. The copycat on the common channel is hence now a relation between moves. This relation is the same as `d-arr` relation."

We are now going to give further details about the validity of the equation that should allow the willing reader to carry on the proof formally.

The first thing to notice is that parallel execution of strategies is in strict relation with Linear Head Reduction [7].

In particular when a Player question move p is copycatted to an Opponent question move t by the parallel composition this means that the variable corresponding to p (correspondence given by the decomposition lemma) is going to be substituted by the subterm corresponding to the Opponent move t (correspondence again given by the decomposition lemma) by Linear Head Reduction.

Hence the `d-arr` relation gives the relation between variables and terms generated by substitution.

[5] Exponential indices are used in [1] to take into account the different occurrences of a variable; in terms of control flow that would allow us to differentiate different calls of the same function, i.e.we would be doing n-cfa instead of 0-cfa. Hyland and Ong [15] use pointers instead of indices to differenciate occurrences of variables; these distinctions are marginal as far as our analysis is concerned.

The function $\mathcal{V}$ then iterates such substitutions until a closure is found. A delicate point at this stage is to prove that the remotion of answers is safe wrt this iteration of substitution. The proof is done by a case analysis.

We say that for a map ϕ from variables to set of terms, $\phi \models M$ if ϕ satisfies the statement of proposition 1; we have then the following subject reduction result:

Proposition 2. *$\mathcal{V} \models M$ and $M \rightarrow_{\mathtt{lhr}} M'$ then $\mathcal{V} \models M'$.*

Proof. Notice first that if $M \rightarrow_{\mathtt{lhr}} M'$ then all variables (free, bound) in M' are also variables in M. Let C be a closure that can be substituted for a variable x in the Linear Head Reduction of M'. Then C is also a closure that can be substituted for a variable x in the Linear Head Reduction of M (because $M \rightarrow_{\mathtt{lhr}} M'$). Hence $C \in \mathcal{V}(x)$, henceforth $\mathcal{V} \models M'$ □

4 k-limited CFA

The objective of this analysis is to identify applications (call sites) at which only a small number of functions may be called (k or less). This information could be used as a basis for an inlining transformation, for example.

Recall that terms are required to be of the form:

$$M_1 \ldots M_n$$

where each M_i is in normal form and the whole term is closed. One consequence of this is that, with the exception of the top-level call site, all applications are of the form:

$$x M_1 \ldots M_r$$

In the standard CFA, the functions that x can evaluate to are computed by $\mathcal{V}$ (which uses $\mathcal{V}'$). We define k-limited CFA by providing an alternative functions $\mathcal{K}$ and $\mathcal{K}'$.

First, we introduce a non-standard set union operator. The k-limited CFA will either associate a (small) set of functions or the special value *many* with a call site. The non-standard set union, $\oplus$, is commutative, associative and satisfies:

$$\begin{aligned} z \oplus y &= \begin{cases} z \cup y & \text{if } \mid z \cup y \mid \leq k \\ \textit{many} & \text{otherwise} \end{cases} \\ z \oplus \textit{many} &= \textit{many} \end{aligned}$$

Given a structure as constructed in Section 2, we define $\mathcal{K}$ on variable labelled vertices of the graph:

$$\mathcal{K}(x) = \bigoplus_{z \in \text{d-arr}(x)} \begin{cases} \{\lambda x.M\} & \text{if } z = [\lambda x.M] \\ \oplus(\mathcal{K}'(\mathcal{K}(y))^{n-\text{cut}}) & \text{if } z = [y M_1 \ldots M_n] \\ \mathcal{K}'(N) \oplus \mathcal{K}'(N') & \text{if } z = [\mathtt{cond}\ M\ N\ N'] \end{cases}$$

where n-cut is as defined earlier and $\mathcal{K}'$ is the obvious extension to sets of the following map:

$$\begin{array}{ll}\mathcal{K}'(\mathit{many}) = \mathit{many} & \\ \mathcal{K}'(M) \quad = \begin{cases} \{\lambda x.N\} & \text{if } M = [\lambda x.N] \\ \mathcal{K}'((\mathcal{K}(y))^{n+r-\text{cut}}) & \text{if } M = [yM_1 \ldots M_n)]^{r-\text{cut}} \end{cases} \end{array}$$

The functions $\mathcal{K}$ and $\mathcal{K}'$ share the same structure as $\mathcal{V}$ and $\mathcal{V}'$. The algorithm is quadratic by similar reasoning to that given for $\mathcal{V}$.

5 Extensions

5.1 Call-by-value

We can force the analysis described so far to be correct for a call-by-value interpreter of the language. All we need to do is to *strictify* terms, i.e. force all bound variables to appear in the body of the term. Hence the strictification[6] of a term $M = \lambda x_1 \ldots x_n.M'$ (noted by $\hat{M}$) is defined as:

$$\lambda x_1 \ldots x_n.\mathtt{cond}(X_1 \mathtt{and} \ldots \mathtt{and} X_n)\hat{M}'\hat{M}'$$

where $\mathtt{and}$ is the boolean conjunction and for x_i of type $A_1 \to \ldots \to A_m \to T$ X_i is $x_i C_1 \ldots C_m$ (C_i a constant of type A_i) if T is the basic type $\mathtt{bool}$ and $\mathtt{iszero}(x_i C_1 \ldots C_m)$ if T is the basic type $\mathtt{int}$. Once the $\mathcal{C}$ map is computed for $\hat{M}$ we get the analysis for the call-by-value evaluation of M by throwing away the dummy closures created passing through the C_i. Note that the strictification is not a translation from call-by-name to call-by-value evaluation but it is sufficient for the purposes of the analysis.

5.2 Adding imperative features

Reynolds proposed a synthesis of functional and imperative programming in [25]; Reynolds' language, *Idealised Algol* (IA), has been given a game semantics by Abramsky and McCusker [2]. In this section we focus on control flow analysis. Data flow analysis would require a more sophisticated treatment of the $\mathtt{new}$ constant [2] (see below).

The basic phrase types of a program are built on top of the basic data types. The phrase types include the type of expressions of a given basic data type, variables of a given basic type and commands:

$$B ::= \mathtt{exp}[X] \mid \mathtt{var}[X] \mid \mathtt{com}$$

General types are constructed in the usual way:

$$T ::= B \mid T \to T$$

6 Strictification can be efficently implemented using pointers.

IA extends PCF with this new type structure and several additional constants. First, there is the degenerate command:

$$\mathtt{skip} : \mathtt{com}$$

Second, we can sequentially compose commands:

$$\mathtt{seq} : \mathtt{com} \to \mathtt{com} \to \mathtt{com}$$

Third, the `new` command introduces local variables:

$$\mathtt{new} : (\mathtt{var}[X] \to \mathtt{com}) \to \mathtt{com}$$

Finally we add an assignment; we require a constant which dereferences a variable and a constant for assignment:

$$\begin{array}{c} \mathtt{deref}_X : \mathtt{var}[X] \to \mathtt{exp}[X] \\ \mathtt{assign}_X : \mathtt{var}[X] \to \mathtt{exp}[X] \to \mathtt{com} \end{array}$$

The abstract strategies for the new constants are shown below:

– `assign` and `seq` are represented as follows:

$$\begin{tikzcd} \mathtt{assign} : \mathtt{var[X]} \to \mathtt{exp[X]} \to \mathtt{com} \arrow[d] & \mathtt{seq} : \mathtt{com} \to \mathtt{com} \to \mathtt{com} \arrow[d] \\ \mathtt{read} : \mathtt{exp[X]} \arrow[d] & \mathtt{run}_1 : \mathtt{com} \arrow[d] \\ \mathtt{write} : \mathtt{var[X]} & \mathtt{run}_2 : \mathtt{com} \end{tikzcd}$$

– `new` and `deref` are

$$\begin{tikzcd} \mathtt{new} : (\mathtt{var[X]} \to \mathtt{com}) \to \mathtt{com} \arrow[d] & \mathtt{deref} : \mathtt{var[X]} \to \mathtt{exp[X]} \arrow[d] \\ \mathtt{block} : \mathtt{var[X]} \to \mathtt{com} \arrow[d] & \mathtt{read} : \mathtt{var[X]} \\ \mathtt{cell} : \mathtt{var[X]} & \end{tikzcd}$$

Following [2] we have given nodes in the strategies suggestive names: so, for example, the strategy for assignment starts by reading the value of the expression and then writes the left hand variable; however what is important is not the name of vertices but their types over which the interaction link algorithm work; the types for the vertices are given by the types of the new constants, so for example in the `new` graph the root has the type of the constant, the vertex `block` has type $\mathtt{var}[X] \to \mathtt{com}$ and the vertex `cell` has type $\mathtt{var}[X]$.

The new types and constants are sufficient to encode the basic imperative language. For example,

$$\texttt{int}\ x; \texttt{while}\ \neg(x = 0)\ \texttt{do}\ x\ :=\ x - 1$$

translates as:

$$\begin{array}{l} \texttt{new}(\lambda x : \texttt{var}[N].Y(\lambda c : \texttt{com}.\ \texttt{cond}(\texttt{iszero}(\texttt{deref}\ x)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{skip} \\ \qquad\qquad\qquad\qquad\qquad\qquad (\texttt{seq}\ (\texttt{assign}\ x\ (\texttt{pred}(\texttt{deref}\ x)))\ c))) \end{array}$$

As shown by Reynolds in [25], all of the basic imperative language (beloved of semantics textbooks) can be translated in this way. Using this encoding, we end back at a functional language. Thus we can apply the techniques of earlier sections.

6 Conclusion

We have presented an approach to control flow analysis of PCF. The analysis is based directly on the game semantics developed by Abramsky, Jagadeesan and Malacaria. We claim the following main advantages for our approach:

- The analysis is derived directly from the semantics – correctness is by construction. We believe that our abstract strategies can be formally presented as a simple abstract interpretation of the AJM games; we are currently developing the details.
- Game semantics can model Idealised Algol, so there is a natural extension of our technique to handle higher-order languages with imperative features. Such features can be accomodated in other semantic frameworks, such as operational semantics, but this usually requires the introduction of explicit state components in the semantics. In contrast, imperative features are modelled by games with a minimal extension to the framework.
- By adding answers to the abstract strategies, we can integrate data flow analyses with CFA in a natural way.

There are a number of extensions to the basic framework that require further work.

1. A relatively straightforward extension of the basic control flow analysis would allow us to emulate Shivers' n-CFA analysis [27]. Essentially this just involves an unfolding of the fixed point graph and a reintroduction of exponential indices on moves (i.e. vertices of the graphs) which would allow us to differentiate between different recursive calls of a function.
2. The `new` constant of [25] allows for the treatment of block structure. A more sophisticated modelling of `new`, which properly accounted for the updating of variables [2], would enable us to treat a broader range of the classical data flow analyses [3]. Our basic technique should be applicable but this requires careful study.

3. Our objective is to build program analysis tools for concurrent, object-oriented languages (such as Java). Reynolds has shown that IA can model classes, objects and methods [24]. We are investigating adding concurrency to IA.
4. The semantic foundations of our approach enforces a strong typing discipline on programs. The algorithm only makes essential use of types in one place – the linking algorithm. It should be possible to adapt the algorithm so that it is applicable to type-free languages.

Acknowledgements

The first author is a UK EPSRC Advanced Research Fellow and both authors are partially supported by UK EPSRC grant GR/L40403 – we are pleased to acknowledge the EPSRC's support. The work has benefitted from discussions with Prahlad Sampath and our project collaborators Samson Abramsky and Guy McCusker.

References

1. Abramsky S., Jagadeesan R. and Malacaria P. Full abstraction for PCF (extended abstract). In *Proc. TACS'94*, LNCS 789, pp 1–15, Springer-Verlag, 1994.
2. Abramsky S. and McCusker G. Linearity, sharing and state: a fully abstract game semantics for Idealised Algol with active expressions. Draft manuscript, 1997.
3. Aho A. V., Sethi R. and Ullman J. D. *Compilers: Principles, Techniques, Tools.* Addison–Wesley, 1986.
4. Banerjee A. A modular, polyvariant, and type-based closure analysis. In *Proc. ICFP'97*, ACM Press, 1997.
5. Bondorf A. *Automatic autoprojection of higher order recursive equations.* Science of Computer Programming, **17**(1-3), pp 3–34, 1991.
6. Cousot P and Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. POPL'77*, pp 238–252, ACM Press, 1977.
7. V. Danos, H. Herbelin and L. Regnier. Game semantics and abstract machines. In *Proc. LICS'96*, IEEE Press, 1996.
8. Girard J.–Y. Towards a Geometry of Interaction. In J. W. Gray and A. Scedrov (eds), *Categories in Computer Science and Logic*, Vol. 92 of *Contemporary Mathematics*, pp 69–108, American Mathematical Society, 1989.
9. Hankin C., Nielson F., Nielson H. R. and Talcott C. Flow analysis for Obliq_0. Draft manuscript, 1997.
10. Hankin C., Nielson F. and Nielson H.R. *Principles of Program Analysis (Volume 1)*, in preparation.
11. Hecht M. S. *Flow Analysis of Computer Programs.* North-Holland, 1977.
12. Heintze N. Set-based analysis of ML programs. In *Proc. LFP'94*, pp 306–317, ACM Press, 1994.

13. Heintze N. and McAllester D. On the Cubic–Bottleneck of Subtyping and Flow Analysis. In *Proc. LICS'97*, IEEE Press, 1997
14. Heintze N. and McAllester D. Linear-time Subtransitive Control Flow Analysis. In *Proc. PLDI'97*, ACM Press 1997.
15. Hyland M. and Ong L. On full abstraction for PCF: I, II and III. 130 pages, ftp-able at `theory.doc.ic.ac.uk` in directory `papers/Ong`, 1994
16. Jagannathan S. and Weeks S. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL'95*, pp 393–407, ACM Press, 1995.
17. Jensen T. P. and Mackie I. Flow analysis in the Geometry of Interaction. In *Proc. ESOP'96*, LNCS 1058, pp 188–203, Springer-Verlag, 1996.
18. Jones N. D. Flow analysis of lambda expressions. In *Proc. ICALP'81*, LNCS 115, pp 114–128, Springer-Verlag, 1981.
19. Milner R. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1-22,1997.
20. Mossin C. *Flow Analysis of typed, higher-order programs*, PhD thesis, DIKU, University of Copenhagen, 1997.
21. Muchnick S. S. and Jones N. D. (eds). *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.
22. Nielson F. and Nielson H. R. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL'97*, pp 332–345, ACM Press, 1997.
23. Palsberg J. Global program analysis in constraint form. In *Proc. CAAP'94*, LNCS 787, pp 276–290, Springer-Verlag, 1994.
24. Reynolds J. C. Syntactic control of interference. In *Proc. POPL'78*, pp 39–46, ACM Press, 1978.
25. Reynolds J. C. The essence of Algol. In J. W. de Bakker and J. C. van Vliet (eds), *Algorithmic Languages*, pp 345–372, North-Holland, 1981.
26. Sestoft P. Replacing function parameters by global variables. In *Proc FPCA'89*, ACM Press, 1989.
27. Shivers O. *Control Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD Thesis, Carnegie–Mellon University, 1991.
28. Yi K. Compile-time detection of uncaught exceptions in Standard ML programs. In *Proc. SAS'94*, LNCS 864, pp 238–254, Springer-Verlag, 1994.
29. Yi K. and Ryu S. Toward a Cost-Effective Estimation of Uncaught Exceptions in SML Programs. In *Proc SAS'97*, LNCS 1302, pp 98–113, Springer Verlag, 1997.