Generalised Flowcharts and Games (Extended Abstract)

Pasquale Malacaria and Chris Hankin

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

Abstract. We introduce a generalization of the classical notion of flowchart for languages with higher order and object-oriented features. These general flowcharts are obtained by an abstraction of the game semantics for Idealized Algol and as such rely on a solid mathematical basis. We demonstrate how charts may be used as the basis for data flow analysis.

1 Introduction

The objective of program analysis is to statically determine some aspect of a program's dynamic behaviour. Such information has traditionally been used in optimising compilers but it also can be used in program verification, providing an alternative to model checking that can deal with infinite state spaces, and formally-based debugging. Many of the traditional approaches to program analysis assume the existence of a control flow graph or flowchart of the program [3]. For first-order imperative languages, the construction of such a graph is relatively trivial. Unfortunately the same is not true for modern programming languages which combine object-oriented, higher-order and concurrency features. The first contribution of this paper is to develop a generalised notion of flowchart which does apply to these languages; our notion is based on an abstraction of game semantics [1]. We also discuss how these generalised flowcharts can be used as a basis for data flow analysis [3].

2 Idealised Algol

Idealised Algol(IA) is a synthesis of functional and imperative programming, originally proposed by Reynolds [12]. The basic types of IA are

$$B ::= \mathtt{Exp}[X] \mid \mathtt{Var}[X] \mid \mathtt{com}$$

where X is a basic type `integer` or `boolean` . General types are constructed in the usual way:

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

The syntax of the language is as follows:

$$x \in \mathit{Var} \qquad c \in \mathit{Const} \qquad e \in \mathit{Exp}$$
$$e ::= x \mid c \mid e_1\ e_2 \mid \lambda x.e$$

Integer constants: The basic integer constants are the numeral zero: $0 : \texttt{Exp}[\texttt{integer}]$ and the successor operation, predecessor operation and test for zero predicate:

$$\texttt{succ}, \texttt{pred} : \texttt{Exp}[\texttt{integer}] \to \texttt{Exp}[\texttt{integer}]$$
$$\texttt{iszero} : \texttt{Exp}[\texttt{integer}] \to \texttt{Exp}[\texttt{boolean}]$$

Boolean constants: The basic boolean constants are true and false and there is a conditional at each type:

$$\texttt{tt}, \texttt{ff} : \texttt{Exp}[\texttt{boolean}] \qquad \texttt{if} : \texttt{Exp}[\texttt{boolean}] \to T \to T \to T$$

Imperative constants: The basic imperative constant is the skip command: $\texttt{skip} : \texttt{com}$ In addition, commands can be sequentially composed using the sequencing operation: $\texttt{seq} : \texttt{com} \to \texttt{com} \to \texttt{com}$ and there are operations for generating local storage, dereferencing variables and assigning variables:

$$\texttt{new} : (\texttt{Var}[X] \to \texttt{com}) \to \texttt{com} \qquad \texttt{deref} : \texttt{Var}[X] \to \texttt{Exp}[X]$$
$$\texttt{assign} : \texttt{Var}[X] \to \texttt{Exp}[X] \to \texttt{com}$$

Finally, there is a fixed point operator at each type: $\texttt{Y} : (T \to T) \to T$. 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 (see later) which allows us to substitute one occurrence of f at the time.

Semantics. The small step operational semantics of IA we will refer to in this paper is the usual one for the constants of the language ([12]) and is *linear head reduction* for the lambda redexes [4]. 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; replace (only) this occurrence of x with a copy of the corresponding argument N (correspondence given by λ binding); if this was the only occurrence of x then erase the original N and the corresponding λx. The semantics is defined in terms of *configurations*; a configuration is either a state, or a pair $< M, \sigma >$ where M is a term and σ is a state. A *redex* is a configuration to which a semantic rule applies; a *normal form* is a configuration to which no rule applies. We will require a *loose* semantics which associates a set of derivation sequences to a program; the set represents the usual operational semantics except that conditionals are interpreted as nondeterministic choice. We write $k \to^* k'$ to denote that the configuration k' is reachable from the configuration k using the loose semantics.

Here is an example of evaluation of an IA program (we use the standard infix notation for the imperative rules and write at each step the operational rule applied):

$$\begin{array}{llll}
< x := 2; y := (\lambda z.z + z)!x, \epsilon > & \to_{\text{seq},:=} & < y := (\lambda z.z + z)!x, [x \mapsto 2] > & \to_\lambda \\
< y := (\lambda z.!x + z)!x, [x \mapsto 2] > & \to_! & < y := (\lambda z.2 + z)!x, [x \mapsto 2] > & \to_\lambda \\
< y := 2 + !x, [x \mapsto 2] > & \to_! & < y := 2 + 2, [x \mapsto 2] > & \to \\
< y := 4, [x \mapsto 2] > & \to_{:=} & < \epsilon, [x \mapsto 2, y \mapsto 4] > &
\end{array}$$

where $!x$ is a shorthand for $\texttt{deref}(x)$. We define the *first occurence of a redex* (FOR in short) as a redex such that none of its subterms has been evaluated (e.g. in the example above $y := (\lambda z.z+z)!x$ is a FOR but $y := 2+2$ isn't). Given a program P the *execution trace* of P is defined as the sequence of FORs in the evaluation of P in the loose semantics. We write $r \leq r'$ if the FOR r precedes r' in the execution trace of P.

3 Object-Oriented Languages in IA

As Reynolds showed [11], Classes can be interpreted in IA by using higher order functions. Formally the statement

$$\texttt{DefineClass}\ C\ \texttt{as}\ \texttt{Decl}; \texttt{Init}; M_1, \ldots M_n$$

where $\texttt{Decl}; \texttt{Init}$ declare local variables and initalise them and the M_i are the methods defined in C can be translated as the term $\hat{C}$ where $\hat{C}$ is

$$\lambda c^{(\mu_1 \times \ldots \times \mu_n) \rightarrow \texttt{com}}.\texttt{Decl}; \texttt{Init}; c < M_1, \ldots M_n >$$

of type $((\mu_1 \times \ldots \times \mu_n) \rightarrow \texttt{com}) \rightarrow \texttt{com}$. Instantiation of classes as:

$$\texttt{newelement}\ x : C\ \texttt{in}\ P \text{ is translated by } \hat{C}(\lambda x^{(\mu_1 \times \ldots \times \mu_n)}.P)$$

A natural extension of Reynold's translation allows us to handle subclasses (with single inheritance). The translation of the statement

$$\texttt{DefineClass}\ C'\ \texttt{Subclassof}\ C\ \texttt{as}\ \texttt{Decl}; \texttt{Init}; M'_1, \ldots M'_l$$

is given by $\hat{C'}$ which is defined by

$$\lambda c^{\mu_1 \times \ldots \times \mu_n} c'^{(\mu_1 \times \ldots \times \mu_n \times \mu'_1 \times \ldots \times \mu'_l) \rightarrow \texttt{com}}.\texttt{Decl}; \texttt{Init}; c' < c.1, \ldots, c.n, M'_1, \ldots M'_l >$$

where $c.i$ is the $i-$th element of the tuple.

Instantiation of subclasses as in:

$$\texttt{newelement}\ y : C' < C\ \texttt{in}\ P \text{ is given by } \hat{C}(\lambda x^{(\mu_1 \times \ldots \times \mu_n)}.\hat{C'}\ x\ (\lambda y^{(\mu'_1 \times \ldots \times \mu'_l)}.P))$$

Overriding is implemented as follows: Suppose we want to override the $i-$th method of C in C' with M'_i. Then $\hat{C'}$ is:

$$\lambda c\ c'.\texttt{Decl}; \texttt{Init}; c' < c.1, \ldots, c.(i-1), M'_i, c.(i+1), \ldots, c.l, M_1, \ldots M_n >$$

where the types of the variables c, c' are as above.

A problem arising when using the translation for subclasses is that we get a type error if we try to implement subsumption. For example if we have defined a procedure $P(x)$ where x is a parameter of type Class C then subsumption should allow us to call $P(k')$ where k' is an instance of the subclass C'. However there is a type mismatch: According to the translation x has type $\mu_1 \times \ldots \times \mu_n$ whereas k' has type $\mu_1 \times \ldots \times \mu_n \times \mu'_1 \times \ldots \times \mu'_l$.

This problem is overcome by extending IA with a notion of subtyping whose base case is $A < A \times B$ and is then extended to arrow types by $A < A', B' < B$ implies $A' \to B' < A \to B$.

As far as our analysis is concerned subsumption is handled as follows: the typechecker associates more than one type to terms according to their subtyping instantiations (in the example above P is going to have the two possible types) specifying which one is appropriate for a particular program point; in the example above at the program point $P(k')$ the appropriate type for P will be $(\mu_1 \times \ldots \times \mu_n \times \mu'_1 \times \ldots \times \mu'_l) \to \texttt{com}$. Given this information the algorithm is then the usual one (i.e. the one for IA without subtyping).

4 The framework

We will be concerned with two player games [1]; we designate the two players by P, for Player, and O, for Opponent. The player represents the program and the opponent represents the environment. Players can make two kinds of moves: Questions and Answers. Formalising this, a game of type A is a triple, (M_A, λ_A, P_A), consisting of a set of moves, a labelling function (which specifies whether a move is a player/opponent question/answer move) and a set of valid positions.

The set of valid positions of a game of type A is a non-empty, prefix closed subset of the set of sequences of moves. In addition, elements of P_A satisfy the following three conditions: every valid position starts with an opponent move; moves alternate between player and opponent; there are at least as many questions as there are answers – this condition is called the *bracketing* condition.

Games correspond to types; the games corresponding to non-basic types are constructed according to the type constructors involved in the type. For example, the game for $A \multimap B$ is constructed from the games for A and B. The moves are the union of the moves from the two component games. The labelling function complements labels in A. The valid positions are constrained such that if we project over either A or B we get a valid position in the respective game, only the player is allowed to switch from A to B and answer moves correspond to the most recent unanswered question.

To model programs (of a particular type) we introduce the notion of *strategy* – a non-empty set of even length valid positions (the even length constraint ensures that any position ends with a P move). We further require that for any strategy, σ:

$$\overline{\sigma} = \sigma \cup \{sa \in P_A | \exists b.sab \in \sigma\}$$

is prefix closed. We can think of strategies as (infinite) trees – the trees may have infinite branches (assuming that answers are drawn from an infinite set) and infinite paths (if we allow fixed points – as we do).

The usual function space, $A \to B$, is constructed as the set of strategies in $(!A \multimap B)$ where $!A$ is the game for A repeated *ad libitum* (see [1] for details). For example, the strategy for a unary function, f, on natural numbers is:

$$\{\varepsilon, q_O q_P\} \cup \{q_O q_P \underline{n}\ \underline{m} \mid f(n) \to^* m\}$$

Both [1] and [6] define categories of games in which the games are objects and strategies are the morphisms. The categories are cartesian closed and thus provide a model for the standard denotational metalanguage.

Application of one program to another is modelled by parallel composition of the corresponding strategies, followed by hiding the interaction [1].

5 Generalized Flowcharts

For sake of simplicity in the following we are going to assume that the term to be analysed has the form, $M_1 \ldots M_n$, where each M_i is in normal form and the whole term is closed[1].

Given a normal form M (not containing the fixed point constant), its tree translation we are going to give is the tree M_0 obtained by collapsing the game interpretation $\mathcal{G}[\![M]\!]$ [2] of M according to the following rule: M_0 is $\mathcal{G}[\![M]\!]$ where justification pointers have been removed and every answer move has been replaced by the symbol $\star$. Notice that M_0 is always finite. If M is the fixed point constant then a graph (with cyclic paths) is required to finitize the game interpretation of M.

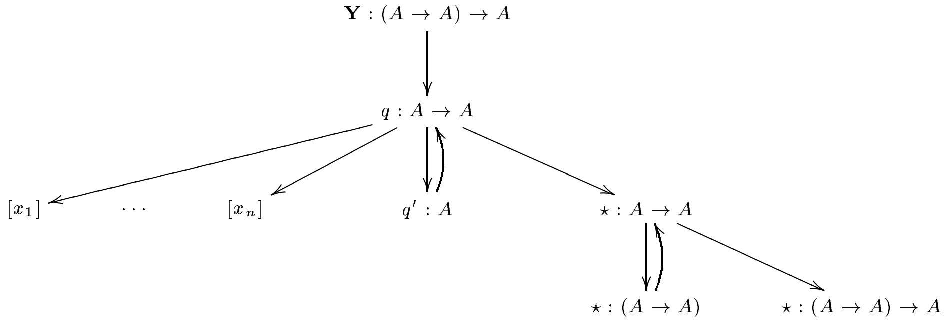
The rest of this section is devoted to explicitily define M_0

- `skip`, `assign`, `seq` and `deref` are represented as follows:

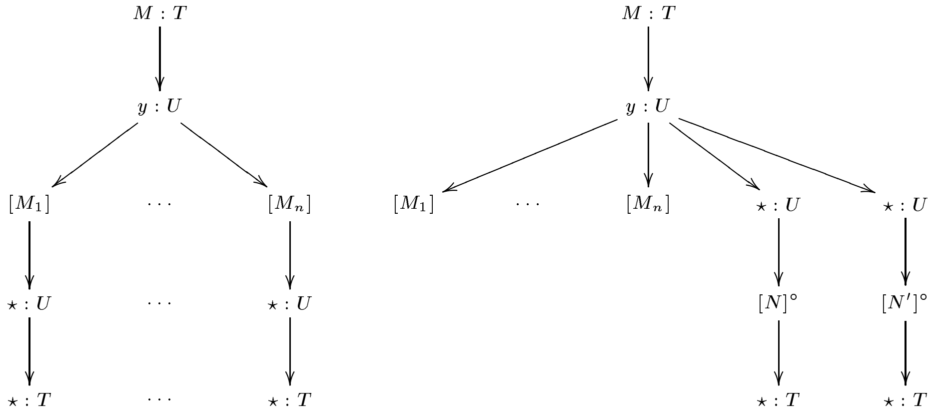
$$\begin{tikzcd}
\texttt{skip} : \texttt{com} \arrow[d] \\
\star : \texttt{com}
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{assign} : \texttt{Exp[X]} \to \texttt{Var[X]} \to \texttt{com} \arrow[d] \\
\texttt{read} : \texttt{Exp[X]} \arrow[d] \\
\star : \texttt{Exp[X]} \arrow[d] \\
\texttt{write} : \texttt{Var[X]} \arrow[d] \\
\star : \texttt{Var[X]} \arrow[d] \\
\star : \texttt{Exp[X]} \to \texttt{Var[X]} \to \texttt{com}
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{seq} : \texttt{com} \to \texttt{com} \to \texttt{com} \arrow[d] \\
\texttt{run}_1 : \texttt{com} \arrow[d] \\
\star : \texttt{com} \arrow[d] \\
\texttt{run}_2 : \texttt{com} \arrow[d] \\
\star : \texttt{com} \arrow[d] \\
\star : \texttt{com} \to \texttt{com} \to \texttt{com}
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{deref} : \texttt{Var[X]} \to \texttt{Exp[X]} \arrow[d] \\
\texttt{read} : \texttt{Exp[X]} \arrow[d] \\
\star : \texttt{Exp[X]} \arrow[d] \\
\star : \texttt{Var[X]} \to \texttt{Exp[X]}
\end{tikzcd}$$

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

[1] This assumption doesn't affect the complexity of the algorithm [8].

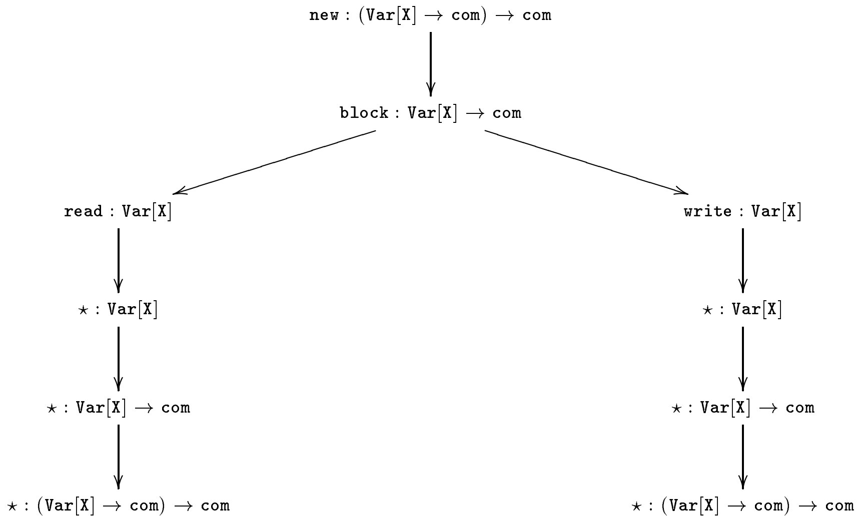


– $M = \lambda x_1 \dots x_m.yM_1 \dots M_n$ and $M = \lambda x_1 \dots x_m.\texttt{cond}\ (yM_1 \dots M_n)\ N\ N'$ are



where T and U are the types of the term M and the variable y and $[H]^\circ$ is the translation of $[H]$ pruned of its root.

– `new` is



The other delta rules succ, pred, iszero are all interpreted by the same tree which is the obvious abstraction of the strategy discussed in the previous section.

In order to handle Object-Oriented features we need to iterpret normal forms of product type. All graphs described so far have a root; products break this rule; for $< a, b >: A \times B$ its translation $[< a, b >]$ is given by the pair of graphs $[a][b]$.

To adjust the setting to cope with multiple roots we stipulate that in all the previous clauses of the translation, edges of the shape

$$\begin{tikzcd} x \arrow[d] \\ [M] \end{tikzcd}$$

are families of edges, one edge for each root of $[M]$.

Projections $\Pi_i : A_1 \times \ldots \times A_n \to A_i$ are interpreted by:

$$\begin{tikzcd} p : A_1 \times \ldots \times A_n \to A_i \arrow[d] \\ q_i : A_i \arrow[d] \\ \star : A_i \arrow[d] \\ \star : A_1 \times \ldots \times A_n \to A_i \end{tikzcd}$$

We will call *variables* the non $\star$ vertices at an even level of the tree and *subterms* the non $\star$ vertices at an odd level of the tree (the root of a tree is at level 1, for the fixpoint graph apply the convention erasing the looping (i.e. the upward) edge). $\star$ vertices are called answers; the ones at an even level of the tree are P-answers and the ones at an odd level are O-answers. This definition comes from the game interpretation of programs where variables (resp. subterms) are interpreted by Player (resp. Opponent) moves.

Interaction links. We are now going to connect the family of graphs $(M_1)_0 \ldots (M_n)_0$ generated by the previous section. These links between graphs of normal forms are called *dotted arrows* and are directed edges from variables to subterms and from P-answers to O-answers. It's enough to describe how links from variables to subterms are created because the ones from P-answers to O-answers are completely symmetric (each variable has associated a unique O-answer and each subterm has associated a unique P-answer).

- Notice first that by definition each variable and subterm in $M_1 \ldots M_n$ has associated a unique occurrence of a type. The difference between type and occurrence is essential in the analysis.
- Let $A_2 \to \ldots \to A_r \to B$ 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, $1 \leq i \leq n$ make the following associations 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 $A'_1 \to \ldots \to A'_m \to B$ and $N_1 \ldots N_h$ are arguments of x (i.e. $xN_1 \ldots N_h$ is a subterm) then N_j is associated with A'_j.
2. Moreover if $N_j = \lambda y_1 \ldots y_k.M''$ and $A'_j = C_1 \to \ldots \to C_h \to B$ Then y_l is associated with C_l.
3. Repeat the same process for each N_j in $1 \leq j \leq h$ going back to the step 1.

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 (in order for twin types to have all the "matching pairs" some η expansions may be needed).

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

We have already noticed that there is a correspondence between question nodes (variables and subterms) and answers nodes. This translates to edges; a *question edge* is an edge leaving from a variable and an *answer edge* is one leaving from a P-answer. Notice that a given question edge has associated a unique answer edge (bracketing condition).

6 Charts and Flowcharts

Given a program define its *chart* as the graph defined in section 5; given nodes a, b in the chart we say that b is *reachable* from a if there is an alternating (dotted/solid arrows) path from a to b.

A *valid path* is a sequence $s_1 \ldots s_n$ such that each s_i is an alternating path and for $j < n$ the target of s_j is a variable node whose answer is the source of s_{j+1}.

A *1-valid path* is a valid path $s_1 \ldots s_n$ starting from the root of the graph with no question edge repeated and for which there exists a (edge) well-bracketed alternating path completion, i.e. there exists $s'_1, \ldots, s'_n$ such that the sequence $s_1 s'_1 \ldots s_n s'_n$ is an alternating path which is edgewise well bracketed.

Given nodes a, b in the chart we say that b is *1-reachable* from a if there is a 1-valid in which a appears before b.

Given a *chart* of a program several kinds of information can be deduced: Given a subset N of nodes of a chart C define a N-*Flowchart* as the preorder $<_1$ on nodes in N defined by:

$$n <_1 n' \text{ iff the node } n' \text{ is 1-reachable from } n \text{ in } C$$

The worst case complexity for building an N-Flowchart is cubic in the size of the program; however this situation is rarely encountered; moreover for restricted languages N-Flowcharts are easier to build; for example the algorithm restricted to imperative programs has linear complexity.

Given a chart, a FOR is identified with the vertex which is the root for the translation of the leftmost term in the redex.

We call a *program* an IA term of type $\sigma_1 \to \ldots \to \sigma_n$ with all σ_i of basic type.

Theorem 1. *Let $P = TT_1 \ldots T_n$ be a (normalized) program and c, c' two nodes in the chart of P corresponding to two FORs of P; the following are equivalent:*

- $c <_1 c'$.
- *In the execution trace of P, $c \leq c'$*

In order to relate our general notion of Flowchart with the classic one proceed as follows. For a first-order program without procedure calls, P, take as subset N the nodes in the chart of P corresponding to the following program points: `cond`, `assign`. Note that we don't need any special node for `while` loops because of the lack of antisymmetry in the definition of preorder. We have then

Corollary 1. *For N and P as above the N-Flowchart of P and the usual Flowchart of P coincide (i.e. they are graph isomorphic).*

The proof of Theorem 1 relies on the relationship between composition in the category of Games and linear head reduction [4].

Our main theorem can be restated as follows (here $\sigma \| \tau$ stands for the parallel composition of σ and τ [1]):

Theorem 2. *Let P and c, c' as in Theorem 1; the following are equivalent:*

- $c <_1 c'$.
- *In $\mathcal{G}[\![T]\!] \| \mathcal{G}[\![T_1]\!] \| \ldots \| \mathcal{G}[\![T_n]\!]$ we have $s m_c s' m_{c'} s''$ for some s, s', s''.*
- *In the execution trace of P, $c \leq c'$*

An informal justification for this result is the following: $\mathcal{G}[\![T]\!] \| \mathcal{G}[\![T_1]\!] \| \ldots \| \mathcal{G}[\![T_n]\!]$ is a play; erasing the justification part of moves we get a relation R between moves in correspondence with the copycat of composition; this is the same relation as the dotted arrow relation in the chart of the program. On the other side the same relation R can be translated (via the decomposition lemma) to a relation between variables and subterms of P which corresponds to linear head reduction. Once this is established the result follows rather easily.

7 Data Flow Analysis

Intra-procedural analysis. Data flow analyses are of two kinds: those that use some properties of the data that is being manipulated by the program and those that track the way in which data is used. An example of the former is constant propagation, whilst an example of the latter is reaching definitions [10]. The former class of analyses have been called first-order, whilst the latter have been called second-order. In order to present first-order analyses based on our approach, we would have to introduce some concrete answer moves into our abstract strategies and we will explore this in a forthcoming paper; however, our current framework is sufficient for second-order analyses.

We have already observed that our abstract strategies are generalised flow charts. From the point of view of data flow analysis, we are interested in certain nodes in the abstract strategy: questions corresponding to the roots of assignments, skips and the tests in conditionals. For each such node, we associate a pair of data flow equations which specify the data flow information present at the entry of the node and at the exit – the computation of this information requires access to the arguments of the nodes. For forward analyses, we require the notion of *immediate predecessor*. We write $n_1 <_i n_2$ if $n_1 < n_2$ and there is a path from n_1 to n_2 with no intervening nodes. The equations associated with a node are then:

$$\begin{aligned} n_\circ &= \textstyle\bigcup_{n' <_i n} n'_\bullet \\ n_\bullet &= f_n(n_\circ) \end{aligned}$$

where $n_\circ$ specifies the data flow values at the entry to the node, $n_\bullet$ specifies the data flow values at the exit and f_n is the *transfer function*. The correctness of these equations follows from the correctness of $<$. The solution of such equations is straightforward using standard techniques such as those discussed in [10]. These solution procedures are independent of the underlying framework and thus the complexity is unaffected by the use of our framework.

Inter-procedural analysis. Given our earlier observations, extending these techniques to interprocedural analyses for first-order procedures is straightforward. We can use state-of-the-art algorithms from the literature. An interesting observation is that the common notion of "interprocedurally valid path" is the bracketing condition on valid positions in Game Semantics [1]. If we restrict attention to the nodes of interest, we immediately have an interprocedural flow graph from the abstract strategy. Once we have the flow graph, the analysis is independent of the underlying framework. For example, we can use the approach of Horwitz, Reps and Sagiv [5] and attain the same complexity that they do.

8 Higher-order Procedures

In the presence of higher-order procedures it is necessary to precede the data flow analysis by a closure analysis (to determine which procedures may be called at a particular call site). In an earlier paper, [8], we have demonstrated that charts may be used as the basis for a simple closure analysis which has the same (cubic) complexity as the state-of-the-art algorithms. Unfortunately, the 0-CFA is rather inaccurate – because it can not differentiate between different calls of a function. Consider the following term: $(\lambda f.(\lambda xy.x)(f(\lambda z.1))(f(\lambda z.2)))(\lambda a.a)$ After normalisation and η-expansion this becomes:

$$(\lambda rf.r\ (\lambda xy.x)(f(\lambda z.1))(f(\lambda z.2)))(\lambda cdeg.c\ d\ e\ g)(\lambda ab.ab)$$

The cache map of [8] effectively maps this term to: $\{\lambda g.1, \lambda g.2\}$. In fact the result of reducing the expression is the first term. The inaccuracy arises because the environment coalesces the bindings for a for the two different calls of f. This is a well-known property of the 0-CFA approach.

A number of solutions to this problem have been proposed in the literature (e.g. [9]). We will briefly consider polynomial k-CFA [7] – the k indicating the degree of differentiation that we can make between calls. The approach involves some form of labelling of call sites in the program – the dotted arrows introduced in our algorithm are also labelled with a string which records the last k calls according to the $<$ ordering. The environment ($\mathcal{V}$) and cache ($\mathcal{C}$) functions become (for conciseness we have omitted the conditional):

$$\mathcal{V}(x,\ell) \quad = \bigcup_{z \in Z} \begin{cases} \{(\lambda x.M, \ell)\} & \text{if } z = [\lambda x.M] \\ \bigcup(\mathcal{V}'((\mathcal{V}(y, \ell : \ell'))^{\text{n-cut}}, \ell : \ell')) & \text{if } z = [y^{\ell'} M_1 \dots M_n] \\ \bigcup(\mathcal{V}'((\mathcal{V}(y, \ell))^{\text{n-cut}}, \ell)) & \text{if } z = [y M_1 \dots M_n] \end{cases}$$
$$\text{where } Z = \{y \mid y \in \text{d-arr}(x, \ell)\}$$

$$\mathcal{V}'(M,\ell) = \begin{cases} \{(\lambda x.N, \ell)\} & \text{if } M = [\lambda x.N] \\ \mathcal{V}'((\mathcal{V}(y, \ell : \ell'))^{\text{n+r-cut}}, \ell : \ell') & \text{if } M = [(y^{\ell'} M_1 \dots M_n)]^{\text{r-cut}} \\ \mathcal{V}'((\mathcal{V}(y, \ell))^{\text{n+r-cut}}, \ell) & \text{if } M = [(y M_1 \dots M_n)]^{\text{r-cut}} \end{cases}$$

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

$$\begin{aligned} \mathcal{C}(\lambda x.M, \ell) &= \{(\lambda x.M, \ell)\} \\ \mathcal{C}(M^{\ell'} M_1 \dots M_r, \ell) &= \mathcal{V}'(\{M\}^{\text{r-cut}}, \ell : \ell') \\ \mathcal{C}(x, \ell) &= \mathcal{V}(x, \ell) \end{aligned}$$

where d-arr(x, ℓ) is the set of links labelled ℓ which emanate from x and : concatenates a new label onto the end of the string ensuring that the length of the string does not exceed k – by dropping labels from the beginning of the string if necessary. The function $\mathcal{V}'$ is an auxiliary function that is used to follow dotted arrows transitively and n-cut removes head lambdas; we have defined these functions to operate on single terms – the extension to sets of terms is in the obvious way.
If we label the term: $(\lambda r f.r\ (\lambda xy.x)(f^1(\lambda z.1))(f^2(\lambda z.2)))^3(\lambda cdeg.c\,d\,e\,g)(\lambda ab.ab)$, a 1-CFA is sufficiently precise to give the accurate answer. The polynomial 1-CFA variant of our algorithm is $O(n^6)$, as are the state-of-the-art algorithms in the literature.

9 Conclusions

We have presented a generalised notion of flow charts. This new notion is sufficiently powerful to enable us to capture control flow information for a variety of modern programming language features, including higher-order procedures, object-orientation and concurrency. In this paper we have concentrated on imperative languages with higher-order procedures, which are sufficiently powerful to encode class-based object-oriented languages. We will consider concurrency in a sequel to this paper. We have shown how this framework can be used as a basis for program analysis; this continues a programme of work started in [8] which mainly considers control flow analysis for PCF. A major advantage of the approach is that the different programming language paradigms are handled in a uniform way within the same abstract game semantics framework. For future work, in addition to considering concurrent languages, we would like to see how the framework might be extended to support first-order data flow analysis.

Acknowledgements

We are grateful to our colleagues from the TCOOL project for their support and encouragement. Thanks also to the UK Engineering and Physical Sciences Research Council for funding TCOOL and supporting the first author through an Advanced Research Fellowship. Finally, we are grateful to Bernhard Steffen for his advice and encouragement.

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. V. Danos, H. Herbelin and L. Regnier. Game semantics and abstract machines. In *Proc. LICS'96*, IEEE Press, 1996.
5. Horwitz S., Reps T. and Sagiv M. Demand Interprocedural Dataflow Analysis. Proc. of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press, 1995
6. 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
7. 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.
8. Malacaria P. and Hankin C. A New Approach to Control Flow Analysis. In *Proc. CC'98*, LNCS 1383, pp 95–108, Springer-Verlag, 1998.
9. 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.
10. Nielson F., Nielson H. R. and Hankin C. *Principles of Program Analysis: Flows and Effects.* to appear, 1999.
11. Reynolds J. C. Syntactic control of interference. In *Proc. POPL'78*, pp 39–46, ACM Press, 1978.
12. 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.