Flow Analysis: Games and Nets

Chris Hankin*, Rajagopal Nagarajan**, and Prahladavaradan Sampath***

SecSafe Document	
Number:	SECSAFE-ICSTM-004
Version:	1.0
Classification:	Public
Status:	Submitted

Abstract. This paper presents a graph-based formulation of control-flow analysis using results from game semantics and proof-nets. Control-flow analysis aims to give a conservative prediction of the flow of control in a program. In our analysis, terms are represented by proof-nets and control-flow analysis amounts to the analysis of computation paths in the proof-net. We focus on a context free analysis known in the literature as 0-CFA, and develop an algorithm for the analysis. The algorithm for 0-CFA performs dynamic transitive closure of a graph that is based on the judgement associated with the proof-net. Correctness of the algorithm relies on the correspondence between proof-nets and certain kinds of strategies in game semantics.

1 Introduction

Twenty years ago Neil Jones published [Jon81] in the 8th International Colloquium on Automata, Languages and Programming. The paper opens with the words:

"*A method is described to extract from an untyped λ-expression information about the sequence of intermediate λ-expressions obtained during its evaluation. ... From a flow analysis viewpoint these results extend existing interprocedural analysis methods to include ... the use of functions both as arguments to other functions and as the results returned by them.*"

The paper goes on to present a technique called "control flow analysis" which results in safe descriptions of the states that an abstract machine reaches in evaluating an expression. The paper was a seminal contribution to the field of control flow analysis; this field was further developed by a number of the members of Neil's TOPPS group at the University of Copenhagen (notably [Ses91,Mos97a,Mos97b]). Moreover, the work has inspired a great deal of work on this topic in other research groups [Shi91,Deu92,Pal94,HM97b,NN97,MH98b] (which represents a very small sample!).

* Imperial College of Science, Technology and Medicine, London SW7 2BZ.
** University of Warwick, Coventry CV4 7AL.
*** Teamphone.com, London W1D 7EQ.

Most approaches in the literature specify control-flow analysis as the solution to a set of (in)equations derived from the syntax of the program. Soundness of the analysis is then demonstrated with respect to an operational semantics; i.e. the derived set of (in)equations describes, for each sub-term, at least the set of abstractions that it evaluates to in the operational semantics of the program. An example of such a soundness result is the proof in [NN97] for a constraint based formulation of control-flow analysis. That paper relates various flavours of control-flow analyses to a structural operational semantics (SOS), and the soundness results for the analysis employs sophisticated co-inductive proof techniques.

More recently, work has been reported on using game semantics to provide a framework for control-flow analysis [MH98b,MH98a,MH99,Sam99]. A feature of the game semantic approach is that control-flow analysis can be specified in a *semantics directed* manner; i.e. the analysis is specified directly in terms of the semantics rather than syntax of a program. This opens up the possibility of exploiting the rich structure of game semantics for specifying analyses. For instance, it might be possible to use the *factorization theorems* of game semantics [AM97,AHM98] to extend program analyses to handle new language features in a modular way.

The approach that we report on in this paper is based on game semantics [AJM94,HO94], and uses proof-nets as representations of λ-terms [Mac94,Sam99]. This is in contrast to earlier analyses [MH98b,MH98a,MH99] based on game semantics, which use a Böhm tree like representation of terms. As a result of using this representation, these analyses rely on pre-processing a program into a special form before analysing them. The use of proof-nets to represent terms in this paper removes the need for such pre-processing of programs.

Proof-nets play an important role in the analysis specified in this paper. The proof-net encoding of λ-terms gives rise to structures that are isomorphic to traditional graph representations of λ-terms. Moreover linear head reduction[1] of λ-terms can be expressed in terms of linear head cut-elimination of proof-nets [MP92,Sam99]. Proof-nets and games are also intimately related in many ways and share a common origin in Linear Logic [Gir87]. Proof-nets can be seen as encoding the strategies of game semantics, thereby providing a bridge between syntax and semantics.

We specify the control-flow analysis directly in terms of game semantics. It is based on the interpretation of *interacting* moves in game semantics as linear head-reduction steps of λ-terms [DHR96,Sam99]. The soundness proof of the analysis relies on the close connection between game semantics on one hand, and the syntax and linear head-reduction of λ-terms on the other hand. The soundness proof can be factored into two stages; first establishing the connection between game semantics and linear head reduction, and then demonstrating the soundness of the analysis with respect to game semantics. Only the second stage of the correctness proof is specific to the analysis — the connection between game

[1] a linearized version of head reduction which makes one substitution at a time.

semantics and linear head-reduction is a general semantic property of games and is expected to be applicable to a number of control-flow based analyses.

We also develop an algorithm for control-flow analysis, based on the game semantics directed specification. The algorithm performs *dynamic-transitive closure* of a *cost-resource* graph. The graph structure is extracted from the proof-net representation of the program, and the edges of the graph are labelled with cost and resource components obtained from the proof-net representation.

As we will see later, the graph structure used in the algorithm is in fact a representation of the type of the program. In this sense our work is related to the type-based approaches of [Mos97b,HM97a]. These approaches perform (explicit or implicit) η-expansion (based on the type-structure) of the program, and then use a simple transitive closure procedure to obtain the analysis. The algorithm we present here does not require an η-expansion step, but on the other hand requires a dynamic-transitive closure procedure.

The worst-case complexity of the algorithm we present for control-flow analysis is $\mathcal{O}(n^3)$, and is in line with the state-of-the-art. We can in fact refine the complexity of the algorithm to $\mathcal{O}(n^2m)$, where m is the size of the (partial) type information required by the algorithm, which itself is bounded above by n. By comparison, the algorithm presented in [Mos97b,HM97a], exhibits a better (quadratic) asymptotic complexity, but has to make the assumption that the size of the type information is linear in the size of the (untyped) term. Our algorithm utilizes type information in a similar manner to [Mos97b,HM97a], but without making any assumptions on the size of the type information.

The significance of the contributions of this paper are:

- The analysis framework using games is extended, using proof-nets, to cope with arbitrary (typed) λ-terms whereas previous work has been restricted to pre-processed λ-terms having a special form [MH98b,MH98a].
- The game semantic framework gives rise to a novel algorithm for control-flow analysis. The algorithm operates on the type structure of the program; this is in contrast to other approaches that operate on the syntactic structure of the program. The worst case complexity of the algorithm is $\mathcal{O}(n^3)$ where the term is of $\mathcal{O}(n)$; this is in line with other algorithms for control-flow analysis.
- The correctness of the control-flow analysis uses only standard semantic properties of games. Therefore there is no need to *instrument* the semantics to add control-flow information.
- The soundness proof can be factored into two stages, only one of which is specific to the analysis. It is conceivable to use the connection between game semantics and linear head-reduction as the basis for a number of other control-flow based analyses.

For the sake of simplicity we only consider terms of simply typed λ-calculus under call-by-name evaluation, but the framework and algorithm are equally applicable to PCF and need not be changed to handle conditionals, recursion etc.

The rest of the paper is organised as follows. Section 2 introduces the basic ideas behind game semantics. Section 3 presents proof-nets as syntactic representation for λ-terms. Section 4 outlines the correspondence between proof-nets, games and linear head-reduction. Section 5 describes how control-flow analysis can be performed in our framework. Section 6 presents an algorithm for the analysis. Section 7 contains concluding remarks.

2 Game Semantics

Game semantics [HO94,AJM94] is a relatively new field of semantics where semantic models of programming languages are built from two player games. Game semantics exhibits the compositional nature of standard denotational semantics while at the same time capturing the intensional nature of operational semantics. This is achieved in game semantics by denoting programs by strategies, which express *potential* for computation. Actual computation is then performed by *interacting* with another strategy (representing the environment). The steps of interaction intuitively represent the steps of an operational semantics. In this section we give a brief overview of some of the concepts of game semantics; we refer the reader to the literature [AJM94,HO94,McC98,Har99] for a more complete introduction to game semantics.

Game semantic models are presented as (two person) games between P (for player) and O (for opponent). Each person takes turns in making moves, which can be interpreted as steps of a computation. Roughly speaking, P models the behaviour of a program while O models the *environment* of the program. Each person (O and P) can make two types of moves: Questions and Answers. A question can be interpreted as *calling* a procedure and an answer can be interpreted as *returning* from the procedure with a *value*. There is also a causality relation, called *enabling*, between moves in the games; for example questions cannot be answered before they are asked. At any point in the game, the *position* in the game refers to the sequence of moves made in the game up to that point. The set of moves in a particular game along with some *ground-rules* like enabling are together referred to as the *arena* of the particular game being played. More formally, an arena for type A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where M_A is the set of moves that can be played in the arena,

$$\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$$

is the O/P labelling function on moves, and $\vdash_A$ is the enabling relation on moves. It is usual to place restrictions on the enabling such that only player moves enable opponent moves (and vice versa), initial moves are always opponent questions ("the environment initiates evaluation") and answer moves are enabled by question moves.

Example 1. The arenas for base types have a very simple structure; for example in the case of booleans, the arena consists of an *initial* Opponent move q and two Player moves true and false. The enabling relation then states that the Player moves cannot be played before the initial Opponent move is played.

$$M_{\mathbb{B}} = \{\mathsf{q}, \mathsf{true}, \mathsf{false}\}$$

and

$$\begin{aligned} \lambda_{\mathbb{B}}(\mathsf{q}) &= \mathsf{OQ} \\ \lambda_{\mathbb{B}}(\mathsf{true}) &= \mathsf{PA} \\ \lambda_{\mathbb{B}}(\mathsf{false}) &= \mathsf{PA} \end{aligned}$$

and

$$\begin{aligned} \mathsf{q} &\vdash_{\mathbb{B}} \mathsf{true} \\ \mathsf{q} &\vdash_{\mathbb{B}} \mathsf{false} \end{aligned}$$

□

Having defined arenas for base types, those for composite types are constructed inductively on the structure of the type. For example, given two arenas, we can combine them to model a *function space*. Given arenas A and B we construct the function space $A{\Rightarrow}B$ by switching the O/P polarity of moves in A. Also, the initial moves in A are now enabled by the initial moves in B; intuitively, we do not want to consult an input unless an output is requested in the game[2].

Definition 1 (Function space). *Given two arenas A and B, we can define the arena $A{\to}B$ as:*

$$\begin{aligned} M_{A\to B} &= M_A + M_B \\ \lambda_{A\to B} &= [\bar{\lambda}_A, \lambda_B] \\ \vdash_{A\to B} &= a \vdash_{A\Rightarrow B} b \textit{ iff } (a \vdash_B b) \vee (a \vdash_A b \wedge a \neq b) \vee (a \in I_B \wedge b \in I_A) \end{aligned}$$

□

Given a program, *prog*, the denotation of the program in game semantics, $[\![prog]\!]$, is a *strategy* for P to play a game – formally this is a set of sequences of moves that conform to certain rules (for example, all of the sequences are of even length – Player has the last move; but see the cited papers for examples of additional possible rules).

Example 2. We illustrate the example of the $[\![iszero]\!]$ strategy, played on the arena $\mathbb{N} \to \mathbb{B}$, that returns true if the input is zero and false if the input is of the form $\mathsf{succ}(n)$. One possible play is shown below

$$\begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{B} \\ & & \mathsf{q}^{\mathsf{OQ}} \\ \mathsf{q}^{\mathsf{PQ}} & & \\ \mathsf{succ}(n)^{\mathsf{OA}} & & \\ & & \mathsf{false}^{\mathsf{PA}} \end{array}$$

The labels on the moves indicate the Player/Opponent and Question/Answer nature of moves. Intuitively, the first move requests the result of the function;

[2] This models call-by-name computation.

the Player responds by requesting an argument — this move is played in the (sub) arena corresponding to the type of the argument. When Opponent supplies an argument ($\mathsf{succ}(0)$ say), the player replies to the initial question with $iszero(\mathsf{succ}(0))$, i.e. false. □

Given a strategy denoting a program, executing the program is represented by pitting the strategy for the program against a strategy that represents the context of the program. This corresponds to providing the arguments, and environment for interpreting the free variables of the program. In game semantics, this pairing of program and context is represented by the *composition* of two strategies, for the program and context respectively. The composition operation sets up an *interaction* between strategies whereby some P-moves in one strategy *interact* with O-moves in the other and vice-versa. This is similar to parallel composition of processes in, for example, CCS [Mil89]. In traditional game semantics it is usual for the resulting interactions to be hidden, analogous to the parallel composition with hiding of CCS. However, for the purposes of control flow analysis, it is important that the interaction be exposed.

Example 3. We illustrate interaction between strategies by considering the composition $([\![0]\!];[\![succ]\!]);[\![iszero]\!]$ of three strategies.

$$
\begin{array}{ccccccccc}
\mathbb{N} & & \mathbb{N} & \xrightarrow{[\![succ]\!]} & \mathbb{N} & & \mathbb{N} & \xrightarrow{[\![iszero]\!]} & \mathbb{B}\\
&&&&&&&& \mathsf{q}^{\mathsf{OQ}}\\
&&&& \mathsf{q}^{\mathsf{OQ}} & \cdots & \mathsf{q}^{\mathsf{PQ}} &&\\
\mathsf{q}^{\mathsf{OQ}} & \cdots & \mathsf{q}^{\mathsf{PQ}} &&&&&&\\
0^{\mathsf{PA}} & \cdots & n^{\mathsf{OA}} &&&&&&\\
&&&& \mathsf{succ}(0)^{\mathsf{PA}} & \cdots & \mathsf{succ}(0)^{\mathsf{OA}} &&\\
&&&&&&&& \mathsf{false}^{\mathsf{PA}}
\end{array}
$$

□

In this example, the dotted lines indicate the interaction of between moves in the arenas for the strategies $[\![succ]\!]$, $[\![iszero]\!]$, and $[\![0]\!]$. Note that interacting moves possess opposite (O/P) polarities. Intuitively this corresponds to the fact that the “output” of *succ* is the “input” of *iszero*.

Formally, a program makes a request to the context by playing a P move in its arena. This move then interacts with an O move in the context-arena; playing the interacting O move passes “control” to the strategy denoting the context. Similarly, any response from the context strategy is returned to the program strategy along with “control”.

Another game semantic notion that will be used in this paper is that of the Player-view (P-view) of a position in a game. The Player-view of a position is defined as a particular subsequence of moves in the position. Strategies where the P-moves made at any position are fully determined by the P-view of the position are of independent interest — they are known as the *innocent* strategies and can be represented as a function from a P-view to a P-move. Innocent strategies have been used for constructing fully abstract models of functional languages [HO94,McC98].

Lying behind our informal presentation of game semantics is a category of call-by-name games. The interaction between player and opponent emulates the typical demand-driven evaluation of interpreters for call-by-name languages.

3 Proof-nets

In this section we introduce proof-nets as representations of λ-terms. Proof-nets also support the notion of a *computation path* through a program; i.e. computation can be viewed as visiting a sequence of nodes in the proof-net representation of a program. In this sense proof-nets are analogous to the higher-order flow graphs of [Mos97a,Mos97b], and to the generalized flow-charts of [MH98a].

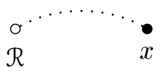
Proof-nets are graphical proof-representations for linear-logic. They were proposed in [Gir87] as a technique of performing multi-conclusion natural-deduction style proofs for linear logic. In this paper, we consider proof-net representations of *intuitionistic propositional* logic. Proof-nets can be considered as representations of terms in the simply-typed λ-calculus via a Curry-Howard correspondence: given a proof-net representation of a term, the *type* of the term can be obtained as the *judgement* proved by the proof-net. In fact, proof-net representations of λ-terms exhibit an isomorphism with a more conventional graph-representation [Wad71] of the terms.

In this paper we draw the proof-nets as forest structures. We associate a positive or negative polarity to the nodes of the forest. The nodes with positive polarity are drawn as $\circ$, and the nodes with negative polarity are drawn as $\bullet$. The axiom links of the proof-net are represented as connections (dotted-lines) between nodes of opposite polarities in the forest. The cut links are represented as connections (solid lines) between the roots of two trees in the forest structure having opposite polarities (cf. Example 5). The proof-nets we consider have a unique root node of positive polarity that is not connected by a cut-link: This corresponds to the root of the term represented by the proof-net. We will use $\mathcal{R}$ to refer to this node. In a traditional graph representation of λ-terms, the node labelled $\mathcal{R}$ corresponds to the unique root of the graph representation of a term.

We refer the reader to the literature [Gir87,Dan90,Lam94,Sam99] for a formal definition of proof-nets; we just give a translation from λ-terms in context to proof-nets. The proof-nets we use in this paper are described in [Sam99], and are simplified versions of general proof-nets, designed with the λ-calculus in mind.

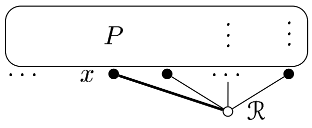
The translation $\mathcal{T}[\![\_]\!]$ from simply-typed λ-terms in context to proof-nets is defined by induction on the structure of a (sequent calculus) derivation of a typed term. The translation presents explicit constructions for contraction and weakening — these are essentially vestiges of the linear logic origins of proof-nets.

Definition 2 (Variable). *We define $\mathcal{T}[\![x : A \vdash x : A]\!]$, where x is of type A, as the proof-net*



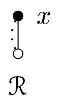
which is an axiom-link in proof-net terminology. Axiom links in proof-nets represent the axioms of the sequent calculus presentation of the proof. □

Definition 3 (Abstraction). *Let* $\mathcal{T}[\![\Gamma, x : A \vdash M : B]\!]{=}P$*, then we define* $\mathcal{T}[\![\Gamma \vdash (\lambda x.M) : A{\Rightarrow}B]\!]$ *from the proof-net* P *by making the* $\bullet$*-node corresponding to the variable* x *an immediate successor of the root node.*

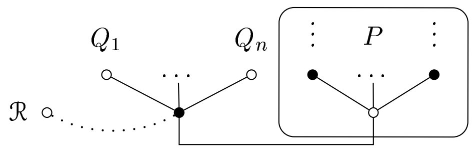


□

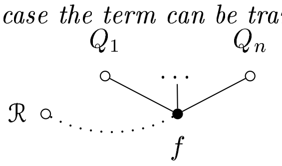
Example 4. Consider the term $\lambda x.x$. We have already seen the representation of the term x as a single axiom link. Abstracting the variable x is now represented by making the $\bullet$-node corresponding to x an immediate successor of the root node ($\mathcal{R}$):



Definition 4 (Application). *Let* $\mathcal{T}[\![\Gamma \vdash M : A_1 \ldots A_n{\Rightarrow}B]\!]{=}P$ *and* $\mathcal{T}[\![\Delta \vdash N_i : A_i]\!]{=}Q_i$*, then we define* $\mathcal{T}[\![\Gamma, \Delta \vdash MN_1 \ldots N_n : B]\!]$ *to be the proof-net*



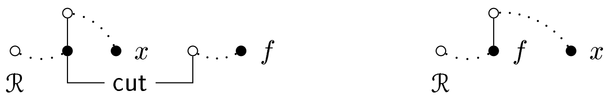
We also give a translation of the term $MN_1 \ldots M_n$ *for the special case where* M *is a* variable*,* f *say. In this case the term can be translated into a proof-net as:*



□

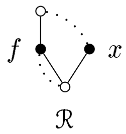
In fact, the special case above can be obtained from the more general translation given above by a series of *axiom-cut elimination* steps on proof-nets. Axiom-cuts are cuts between two proof-nets where at least one of them is a simple axiom link. In general, elimination of cuts in a proof-net is considered to be *computation*. However elimination of Axiom-cuts is considered to be mere bureaucratic rearrangement of the structure of the proof-net, and no computational content is ascribed to these cut-elimination steps [Sam99].

Example 5. Consider the term $\lambda f, x.f(x)$. We have already seen how the variable occurrences f and x can be translated into axiom links. The translation of application terms gives the proof-net (a) below; and the translation for the special case where the operator is a variable occurrence gives the proof-net (b) below. Note the presence of the cut-link in the first translation; this cut-link is elided in the second translation as the cut is an axiom-cut and has no computational significance.

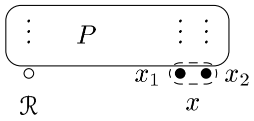


(a) (b)

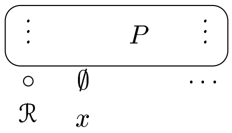
Now applying the translation for abstraction, the translation of the term is given as:



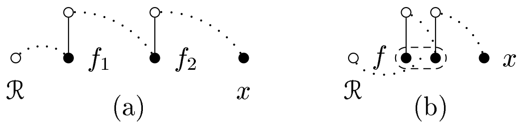
Definition 5 (Contraction and Weakening). *Let $\mathcal{T}[\![\Gamma, x_1 : A, x_2 : A \vdash M : B]\!]=P$, then the contraction of variables x_1 and x_2 as x, $\mathcal{T}[\![\Gamma, x : A \vdash M : B]\!]$is given by the following proof-net where the $\bullet$-nodes corresponding to x_1 and x_2 are combined into a single set, representing x.*



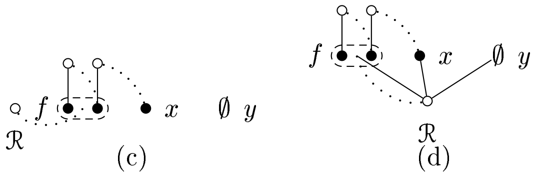
Let $\mathcal{T}[\![\Gamma \vdash M : B]\!]=P$, then we define $\mathcal{T}[\![\Gamma, x : A \vdash M : B]\!]$ where x does not occur in Γ as the following proof-net where x is represented by the empty set, $\emptyset$.



Example 6. Consider the term $\lambda f, x, y.f(f(x))$. This term abstracts the variable y that does not occur in the body, and also contains two occurrences of the variable f. Let us first translate the term $f_1(f_2(x))$, where f_1 and f_2 are distinct. This corresponds to the translation $[\![f_1 : A \to A, f_2 : A \to A, x : A \vdash f_1(f_2(x)) : A]\!]$ and is given by the proof-net (a) below. Now applying contraction to the variable occurrences f_1 and f_2 gives the proof-net (b) below. This corresponds to the translation $[\![f : A \to A, x : A \vdash f(f(x)) : A]\!]$.



Applying weakening for the variable y gives the proof-net (c) and finally applying abstraction for the variables f, x, and y gives the proof-net (d).



Lemma 1. *Given a simply-typed λ-term, the translation of the term using $\mathcal{T}[\![\_]\!]$ results in a proof-net; moreover this proof-net is isomorphic to a traditional graph representation of the λ-term [Wad71].*

Proof. By induction on the structure of the λ-term. The isomorphism with traditional graph-representation of λ-terms maps sequences of abstraction nodes to $\circ$-nodes and sequence of application nodes to $\bullet$-nodes. In the rest of the paper we will refer to terms and proof-nets interchangeably in light of the isomorphism between them. $\square$

The isomorphism between proof-nets and λ-terms gives rise to a correspondence between nodes in the proof-net and pieces of syntax of the λ-term. In the inductive translation of a λ-term, T, every $\circ$-node, a, in the proof-net is the root ($\mathcal{R}$) of the translation of some sub-term, T', of T. The node a in the proof-net then corresponds to the sub-term T'. A similar correspondence can be established for $\bullet$-nodes and variable occurrences. Loosely, each $\bullet$-node represents an *occurrence* of a variable and each $\circ$-node in the proof-net represents a *sub-term.*

Given the proof-net representation of a term, the type of the term can be obtained as the judgement proved by the proof-net. The judgement itself can be obtained by simply considering the forest structure of the proof-net and *forgetting* the contractions and weakenings occurring in the proof net. The resulting forest structure represents the intuitionistic sequent proved by the proof-net: the unique tree with $\circ$-polarity root represents the unique conclusion of an intuitionistic sequent.

The judgement proved by the proof-net is the type of the term represented by the proof-net; this can be seen as a manifestation of the Curry-Howard correspondence. Note that the judgement obtained from the proof-net contains pairs of cut-formulae. By including the pairs of cut-formulae in the type of the term, we obtain a notion of the *intensional type* of a term. Therefore the intensional type of a term is a judgement of the form $\vdash \Gamma; [\cdots]$; where the square brackets enclose a list of pairs of cut-formulae. Intuitively, the intensional type of the term captures some *internal* structure of the term in the form of the cut-formulae.

There is a natural map from the nodes of the proof-net to the nodes in the forest representation of its intensional type, obtained by *forgetting* the effects of weakening and contraction in the proof-net. We will refer to this mapping in the rest of the paper as $\rho_{\mathcal{N},\vdash\Gamma}$, where $\mathcal{N}$ stands for the proof-net and $\vdash \Gamma$ stands for the intensional type of $\mathcal{N}$. Where the context is clear, we will refer to this mapping as simply ρ.

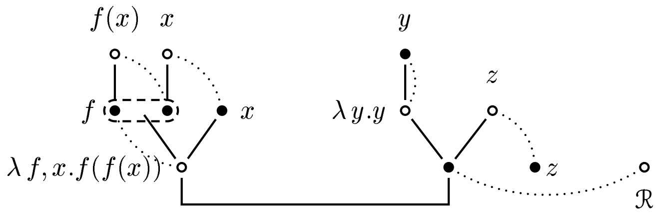
Example 7. We construct the proof-net representation of the term

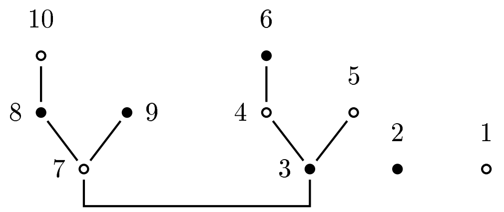
$$(\lambda f, x.f(fx))(\lambda y.y)z$$

We also give the intensional type consisting of the judgement proved by the proof-net along with pairs of cut formulae. Note that the forest structure of the intensional type represents the sequent

$$\vdash \alpha^2 \Rightarrow \alpha^1; [(\alpha^{10} \Rightarrow \alpha^8) \Rightarrow \alpha^9 \Rightarrow \alpha^7 \ \mathsf{cut}\ (\alpha^6 \Rightarrow \alpha^4) \Rightarrow \alpha^5 \Rightarrow \alpha^3]$$

where the superscripts correspond to the labelling on the nodes in the forest representation of the intensional type.





□

We can use the proof-net representation of a term to define a reduction strategy for λ-terms — *linear head-reduction.* This is based on linear head cut-elimination of proof-nets [MP92]. Linear head-reduction, follows the same normalizing strategy as head-reduction, but substitutes for only the *head-occurrence* of the head-variable; it is therefore a *hyper-lazy* evaluation strategy. Linear head-reduction has a very simple characterisation in terms of proof-nets. The linearisation step *separates* the occurrence of the head-variable from the other occurrences, and the cut-elimination step performs the *computation*; i.e. substitution. If we interpret the cut links in the proof-nets as encoding substitutions, then the linearisation step can be interpreted as the reduction

$$(\lambda x_1 \ldots x_m.yE(y))[y \mapsto t] \longrightarrow (\lambda x_1 \ldots x_m.y_1E(y_2))[y_1 \mapsto t][y_2 \mapsto t]$$

where $E(y)$ stands for any expression which possibly contains (non-head) occurrences of y. The cut-elimination step can be interpreted as the reduction

$$\lambda x_1 \ldots x_m.xt_1 \ldots t_n[x \mapsto \lambda y_1 \ldots y_n.t] \longrightarrow \lambda x_1 \ldots x_m.t[y_1 \mapsto t_1 \ldots y_n \mapsto t_n]$$

In this case linear head-reduction is said to *visit*, in sequence, the nodes in the proof-net corresponding to the variable occurrence x (a $\bullet$-node) and the term $\lambda y_1 \ldots y_n.t$ (a $\circ$-node). The linear head-reduction procedure shown above has the attributes of an abstract-machine; for example substitution is performed on occurrences of variables rather than on all occurrences at once (the linearisation step). In this sense the reduction step described above is similar to a *stack-free* Krivine machine.

4 Proof-nets and games

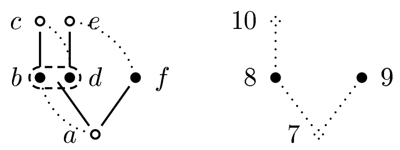
Following [Lam96,Sam98,Sam99], we can view proof-nets as encodings of innocent strategies. The moves in the game are the nodes in the proof-net, modulo the natural map ρ from the proof-net to its intensional type. The P-moves are the $\bullet$-nodes and the O-moves are the $\circ$-nodes in the proof-net(intensional type). Now given the correspondence between proof-nets and λ-calculus presented in Sec. 3, P-moves correspond to occurrences of variables and O-moves correspond to sub-terms.

Given a proof-net $\mathcal{N}$, we can construct a strategy, $\sigma_{\mathcal{N}}$, corresponding to the game semantic denotation of the proof-net by tracing paths in the proof-net ([Sam99, Theorem 4.23]). The basic idea is to obtain a sequence of alternating $\circ$, and $\bullet$ nodes by

- starting with the root ($\mathcal{R}$) of the proof-net
- following axiom-links (dotted lines) for moving to a $\bullet$ node from a $\circ$ node
- and only visiting a $\circ$ node whose predecessor in the forest structure of the proof-net has already been visited.

We can view these paths as *paths of computation* through the proof-net.

Example 8. Consider the term $\lambda f, x.f(f(x))$. The proof-net representation and the intensional type (extracted from Example 7) are given below. Here the nodes of the proof-net are labelled for clarity.



The map ρ from proof-net nodes to nodes in the intensional type is given as

$$\rho = \{\ a \mapsto 7, b \mapsto 8, d \mapsto 8, c \mapsto 10, e \mapsto 10, f \mapsto 9\ \}$$

The paths traced through the proof-net corresponding to game semantics are

$$\{\ a.b, a.b.c.d, a.b.c.d.e.f\ \}$$

These paths, via ρ, give rise to the set of positons:

$$\{\ 7.8, 7.8.10.8, 7.8.10.8.10.9\ \}$$

that fully determines the strategy, $[\![\lambda f, x.f(f(x))]\!]$, on the arena corresponding to the intensional type. $\Box$

In the above interpretation of proof-nets as strategies, the *arena* of the game is the intensional type of the term. However, the proof-net and therefore the intensional type could also include cut-links. A cut-link between two trees in the intensional type identifies pairs of *interacting moves* in the corresponding arena. Two nodes (moves) in the intensional type (arena) *interact* with one another if they are the "dual" nodes, across the cut, of a cut-formula. We also constrain the paths traced throught the proof-net by specifying that whenever a $\bullet$-node is reached that has a "dual" $\circ$-node across a cut, the next node to be visited is this "dual" node. This in fact corresponds to a natural notion of execution in game semantics that we outline below.

Game semantics has a natural notion of *execution* of the semantics. We start off the game with the initial O-move. The strategy then plays a P-move in response. This P-move may then *interact* (cf. Example 3) with some O-move in the arena and O plays this move. This is then followed by the strategy's response to this move and so on, until a P-move is played that does not interact with any other O-move. The trace produced by *executing* a strategy has the form

$$\circ \quad \bullet \bowtie \circ \quad \bullet \bowtie \circ \quad \ldots \quad \bullet \bowtie \circ \quad \bullet \bowtie \circ \quad \bullet$$

where we use $\circ$ to represent an O-move and $\bullet$ to represent a P-move, and where $\bowtie$ represents an interaction between a P and an O-move. The position constructed by *executing* the strategy as above exhibits a close connection with linear-head reduction of λ-terms.

Each pair of *interacting* P and O moves in the execution of a strategy corresponds to a single step of linear head-reduction ([Sam99, Theorem 5.14]). In terms of the syntax, each interacting pair of moves in the execution of a strategy can be interpreted as the *substitution* of a sub-term for an occurrence of a variable. It is this interpretation of *execution of games*, and its connection with linear head-reduction that forms the basis of control-flow analysis of a functional language using game semantics.

5 Control-flow analysis using games

Control-flow analysis (CFA) is a generic term for a range of analyses investigating the dynamics of executing a program. By control-flow, we mean that we are not interested in the flow of data in a program; rather, we are interested in how *control* flows from one function to another during the execution of a program. There are a number of characterisations of control-flow for higher-order functional languages [Shi91,Ses91,Deu92,Jon81,NN97]. We follow the approach to control-flow used in [Ses91]; this approach is also known as *closure analysis* and approximates the set of abstractions that can be bound to a given variable.

In this paper, we abstract the execution trace of game semantics to perform a variant of control-flow analysis generally known as 0-CFA. 0-CFA analysis is essentially a context-free analysis where we do not differentiate between different occurrences of variables. Therefore in the framework of game semantics we can ignore the differences between different occurrence of P-moves. Given a variable in the term, it corresponds to a P-move (α say) in the game semantics of the term. 0-CFA for the variable can now be formalised as follows:

> Capture the set of terms corresponding to all occurrences of O-moves that *interact* with any occurrence of the move α in the execution trace of game semantics.

6 Algorithm for closure analysis

The 0-CFA analysis computes a mapping, $\mathsf{0CFA}$, from variables to the set of *abstractions* in the term that get substituted for the variable. Note in particular that the range of $\mathsf{0CFA}$ contains only abstraction sub-terms.

The idea of the analysis itself is quite simple. The first step is to translate the term to be analysed into its proof-net representation. We can then obtain the intensional type of the the proof-net from the forest structure of the proof-net (cf. Sec. 3). We also compute the map ρ from the proof-net nodes to the intensional type. This mapping basically gives the move in the arena that corresponds to a given variable/sub-term.

Given a variable, x, the analysis first computes the P-move, $\rho(x)$, corresponding to the variable. The analysis then requires the computation of the O-move that interacts with $\rho(x)$. The interaction function, which is named $\bowtie$, can be easily computed from the cut-formulae in the forest structure of the intensional type: given a P-move, α, the O-move interacting with it (if any) is the 'dual' node across a cut link in the forest structure of the intensional type. The control-flow analysis then returns the set of sub-terms that are mapped by ρ to this interacting O-move; i.e.

$$\mathsf{0CFA}(x) = (\rho^{-1} \circ \bowtie \circ \rho)(x) \ .$$

Example 9. Consider the analysis of the term $(\lambda f, x.f(fx))(\lambda y.y)z$, referring to the proof-net representation in Example 7. We have already labelled the nodes in the intensional type.

Now given a variable, f say, we have $\rho(f) = 8$. Now from the intensional type, we can see that $\bowtie(8) = 4$ and finally $\rho^{-1}(4) = \{ \lambda y.y \}$. Therefore, we can conclude that $\mathsf{0CFA}(f) = \{ \lambda y.y \}$.

Similarly, for the variables x and y, we have $\rho(x) = 9$ and $\rho(y) = 6$. Also, from the intensional type, $\bowtie(9) = 5$ and $\bowtie(6) = 10$. Finally, $\rho^{-1}(5) = \{z\}$, and $\rho^{-1}(10) = \{f(x), x\}$. Therefore, we can conclude that $\mathsf{0CFA}(x) = \emptyset = \mathsf{0CFA}(y)$ since neither of these sets contain abstraction terms. $\square$

It should be emphasised that the crux of the analysis is the computation of the interaction function, $\bowtie$, on the intensional type of the proof-net. The actual term is mainly used to compute the *type assignment* ρ, and its inverse ρ^{-1}, which are required for the *read-back* of the analysis. Computing the $\bowtie$ relation effectively performs the same task as the *closing-rules* of [Mos97a] and the conditional constraints of [Pal94], but 0CFA performs the computation on the type structure (intensional type) rather than on the term (proof-net).

In general, the procedure given above is not sufficient to compute 0-CFA. Given a variable x, of higher type, $\bowtie(\rho(x)) = a$ is a $\circ$-node in the intensional type. In general, the sub-term given by $\rho^{-1}(a)$ may not be in the suitable form of an abstraction: $\rho^{-1}(a)$ may be an application term (of higher-order type). In this case we need to perform a step of transitive closure, basically traversing computation paths, to obtain the set of abstraction sub-terms that $\rho^{-1}(a)$ can evaluate to. Transitive closure itself may require new computation paths to be added corresponding to the closing rules of [Mos97a]; this can be represented as a dynamic component of transitive closure.

Let us first consider the case when $a' = \rho^{-1}(a)$ is an application sub-term. Then, in general, the term corresponding to a' is of the form

$$y M_1 \dots M_m$$

Let the node in the proof-net corresponding to the head-variable y be b'. From the translation of terms to proof-nets (Sec. 3), b' is connected by an *axiom link* to a' in the proof-net. Then if the O-move a is played in the execution trace, the strategy will play the P-move $\rho(b') = b$. Now let the strategy play the move $\rho(c') = c$ *interacting* with the move b. This situation is represented as

$$\begin{array}{cccc} \dots & \circ & \bullet \bowtie \circ & \dots \\ & a & b \bowtie c & \end{array}$$

The sub-term corresponding to the node c' in the proof-net has the general-form

$$\lambda y_1 \dots y_n.M$$

Then according to the interpretation of interacting moves, $\lambda y_1 \dots y_n.M$ is substituted for the head-occurrence y. If $n > m$, then the resulting term after reduction is an abstraction of the form

$$\lambda y_{(m+1)} \dots y_n.M$$

and we can conclude that the variable x is bound to this abstraction. Otherwise, if $n \leq m$, then we have to perform the above steps again, starting with the node c', until we reach an abstraction. This procedure is effectively the same as the procedure in [MH98b] to follow computation paths.

This procedure can be rephrased as a graph transitive-closure. The nodes of the graph are the nodes of the *intensional type*; and the edges of the graph are *projections* onto the intensional type, of computation paths in the proof-net. In the above example, there would be an edge from the node a to the node c, indicating that the closure with body a' evaluates to a closure with body c'.

We also need to capture the information concerning the number of abstractions or applications in a term. This can be done by associating a *cost* and *resource* to each edge. Given an edge from a to c, as above, the cost of the edge is the number of *abstractions* in the proof-net node a', and the resource of the edge is the number of *applications* in the node b' in the proof-net (corresponding to the head variable y above). Therefore, in the above example we would have an edge $a \xrightarrow{(0,m)} b$. A similar procedure is carried out for edges between $\bullet$-nodes in the intensional type. Intuitively, the cost and resource information is required to identify sub-terms of interest; since we are considering n-ary abstractions and applications, the cost and resource indicate the sub-term containing "cost" number of abstractions or "resource" number of applications. Effectively, cost-resource (CR) edges emulate the use of *labels* to identify sub-terms.

We can now compose two CR edges,

$$a \xrightarrow{(c_1,r_1)} b \xrightarrow{(c_2,r_2)} c$$

to give an edge $a \xrightarrow{(c,r)} b$ where

$$\begin{array}{ll} (c,r) = (c_1, (r_1 - c_2) + r_2) & \text{if } r_1 > c_2 \\ (c,r) = (c_1 + (c_2 - r_1), r_2) & \text{otherwise} \end{array}$$

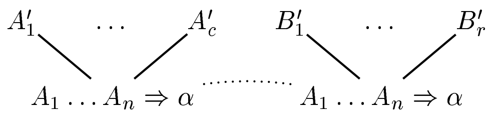
Transitive closure is performed using this definition of composition. The definition of CR edges and the operation of composition bear a remarkable resemblance to the KSL algorithm of [BKKS87]; the cost and resource correspond to the 'K' and 'L' components respectively and the composition operation is a simplified version of the $\oplus$ operation on KSL triples.

Now if there is an edge $a \xrightarrow{(0,r)} b$, then we can conclude that there is a *computation path* from $\rho^{-1}(a) = a'$ to $\rho^{-1}(b) = b'$. In particular, we can conclude that a closure with body a' *may* compute to a closure with body $\mathsf{cut}(r, b')$, where

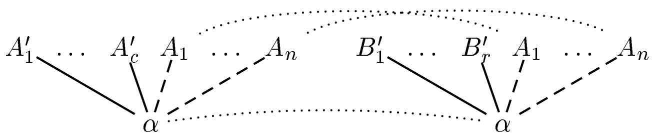
$$\mathsf{cut}(r, \lambda y_1 \ldots y_n.M) = \lambda y_{(r+1)} \ldots y_n.M$$

if $n \geq r$ and is undefined otherwise. This is essentially *read-back* of the analysis. Therefore, given a variable x, such that $\bowtie (\rho(x)) = a$, if $\rho^{-1}(a) = a'$ is an application sub-term, we need to compute all the *zero-cost* paths from a to obtain the abstractions required for the analysis of x.

As we mentioned earlier, new CR edges computed during transitive closure may require additional edges to be added dynamically. These dynamically computed edges correspond to performing η-expansion on the term. Consider an axiom link between nodes a and b of type $A_1 \ldots A_n \Rightarrow \alpha$



The root of the trees (of type $A_1 \dots A_n \Rightarrow \alpha$) can now be expanded and we can infer axiom links between the nodes $A_1 \dots A_n$ as



This corresponds to the η-expansion of the term

$$\lambda x_1 \dots x_c \,.\, y e_1 \dots e_r$$

where $y e_1 \dots e_r$ is of type $A_1 \dots A_n \Rightarrow \alpha$, as

$$\lambda x_1 \dots x_c x_1 \dots x_n \,.\, y e_1 \dots e_r x_1 \dots x_n$$

These inferred axiom links dynamically give rise to CR edges. If there is a CR edge $x \xrightarrow{(c,r)} y$ and suppose z is the mth node immediately above x, where $m > c$, then following the above diagram we add an edge between nodes z and w where w is the $((m-c)+r)th$ node immediately above y. We also set the cost and resource of this edge to be 0; i.e. $z \xrightarrow{(0,0)} w$.

We can now perform 0-CFA as follows:

- Encode terms into proof-nets.
- Compute the intensional type of the proof-net, and the mappings ρ and ρ^{-1}.
- Compute the interaction relation $\bowtie$.
- Project the axiom links of the proof-net onto the intensional type as CR edges.
- Perform dynamic transitive closure (DT), using a standard work-list algorithm [NNH99].
- Read back the analysis. This roughly corresponds to the function

  $$\mathsf{0CFA} = \mathsf{cut} \circ \rho^{-1} \circ \mathsf{DT} \circ \bowtie \circ \rho$$

  Here DT computes the CR edges from a given node, a, by performing dynamic transitive closure. The ρ^{-1} operation considers nodes (b) connected to a by *zero*-cost edges ($a \xrightarrow{(0,r)} b$); and the cut operation removes r number of leading λs from $\rho^{-1}(b)$.

Theorem 1 (Soundness of analysis). *The algorithm for performing 0-CFA analysis is sound with respect to linear head-reduction.*

Proof. This theorem essentially follows from the fact that the dynamic transitive closure only follows paths that are specified by game semantics; i.e. each step of the dynamic transitive closure is sound with respect to the game semantics [Sam99, Theorem 7.6]. □

We note that the soundness proof above only captures soundness of the algorithm with respect to game semantics. The soundness with respect to linear head-reduction follows from the connection between game semantics and linear head-reduction ([Sam99, Theorem 5.14]), which is a general semantic property of game semantics.

Example 10. We illustrate the algorithm on the term $(\lambda\, x.x)(\lambda\, y.y)(\lambda\, z.z)$. Note that this term is not fully η-expanded; variable x of $\lambda\, x.x$ is of higher-order type. Similarly with $\lambda\, y.y$. Fig. 1 gives the proof-net representation and intensional type for this term. The CR edges obtained by projecting the axiom-links of the

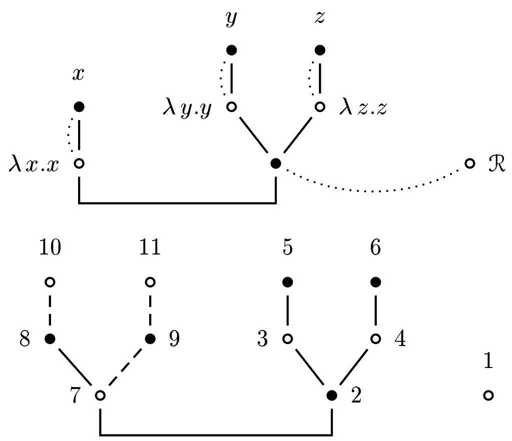


Fig. 1. Proof-net representation of $(\lambda\, x.x)(\lambda\, y.y)(\lambda\, z.z)$

proof-net are

$$\begin{array}{llll} 7 \xrightarrow{(1,0)} 3 & 3 \xrightarrow{(1,0)} 10 & 4 \xrightarrow{(1,0)} 11 & 1 \xrightarrow{(0,2)} 7 \\ 8 \xrightarrow{(0,1)} 2 & 5 \xrightarrow{(0,1)} 8 & 6 \xrightarrow{(0,1)} 9 & \end{array}$$

Performing dynamic-transitive closure on these edges gives rise to the edges

$$\begin{array}{llll} 9 \xrightarrow{(0,0)} 5 & 1 \xrightarrow{(0,1)} 3 & 7 \xrightarrow{(2,0)} 10 & 10 \xrightarrow{(0,0)} 4 \\ 5 \xrightarrow{(0,2)} 2 & 9 \xrightarrow{(0,1)} 8 & 1 \xrightarrow{(0,0)} 10 & 3 \xrightarrow{(1,0)} 4 \\ 7 \xrightarrow{(2,0)} 4 & 9 \xrightarrow{(0,2)} 2 & 1 \xrightarrow{(0,0)} 4 & \end{array}$$

Now computing the function $\mathsf{DT} \circ \bowtie \circ \rho$ for the variables x, y and z gives us the mapping

$$\begin{aligned} x &\mapsto \{\, 3, \xrightarrow{(1,0)} 4, \xrightarrow{(1,0)} 10 \,\} \\ y &\mapsto \{\, 10, \xrightarrow{(0,0)} 4 \,\} \\ z &\mapsto \{\, 11 \,\} \end{aligned}$$

Now performing $\mathsf{cut} \circ \rho^{-1}$ for the zero-cost reachable nodes in the sets, we obtain the analysis

$$\begin{aligned} x &\mapsto \{\, \lambda y.y \,\} \\ y &\mapsto \{\, \lambda z.z \,\} \\ z &\mapsto \emptyset \end{aligned}$$

□

Complexity The algorithm for 0-CFA is computed by a dynamic-transitive closure on the intensional-type structure of the term. The complexity of the analysis is $\mathcal{O}(n^3)$ in the size of the term. Intuitively, the algorithm considers $\mathcal{O}(n^2)$ number of edges[3] and each step of the dynamic-transitive closure performs $\mathcal{O}(n)$ amount of work; this is comparable with similar results in the literature [Pal94,MH98b].

Our algorithm is also related to the type-based algorithms developed in [Mos97b,HM97a]. These approaches perform (explicit or implicit) η-expansion (based on the type-structure) of the program, and then use a simple transitive closure procedure to obtain the analysis. The algorithm we present here does not require an η-expansion step, but on the other hand requires a dynamic-transitive closure procedure. As a result of this tradeoff, the algorithm presented here does not have some of the pleasant properties of the algorithms in [Mos97b,HM97a]. For example, it is not possible to obtain control-flow information for individual variables independently of the rest of the analysis; this is a result of performing dynamic-transitive closure as opposed to simple transitive closure.

The runtime complexity of our algorithm is $\mathcal{O}(n^2 m)$, where n is the size of the program and m is the size of the partial intensional type required by the algorithm, and where m is bounded above by n in the worst case. This is in line with state-of-the-art 0-CFA algorithms that exhibit a $\mathcal{O}(n^3)$ complexity. Compared to the type-based 0-CFA algorithms that exhibit a quadratic complexity [Mos97b,HM97a], our algorithm does not make the assumption that the size of the type-annotated term is linear in the size of the untyped term[4].

[3] The $\mathcal{O}(n^2)$ figure is not obvious from the formulation of the CR edges in the algorithm, but it can be shown that the number of edges considered in the algorithm can be bounded by $\mathcal{O}(n^2)$ [Sam99].

[4] In general, the size of the type-annotated term is arbitrarily larger than the size of the untyped term

Interpretation as constraints The algorithm for 0-CFA given above has a close relation (as expected) with the constraint based formulations of the analysis [Pal94]. Each of the CR edges considered in the algorithm can be interpreted as an inclusion constraint on the λ-term. Each edge, $a \xrightarrow{(c,r)} b$, between $\circ$-nodes a and b, can be interpreted as a constraint

$$\mathsf{cut}(r, \rho^{-1}(b)) \subseteq \mathsf{cut}(c, \rho^{-1}(a))$$

and each edge $l \xrightarrow{(c,r)} m$, between $\bullet$-nodes l and m, can be interpreted as a constraint

$$\mathsf{take}(c, \rho^{-1}(l)) \subseteq \mathsf{take}(r, \rho^{-1}(m))$$

where the function take is defined as

$$\mathsf{take}(r, xM_1 \ldots M_n) = xM_1 \ldots M_r$$

if $n \geq r$ and is undefined otherwise.

7 Conclusion and Future work

We have presented a technique of performing control-flow analysis for a functional language. The analysis is based on a game semantic model of the language. We have also presented an algorithm for the analysis, using proof-nets as the syntactic representation of programs. The algorithm presented has state-of-the-art performance, and the proof of correctness essentially follows from the game semantic model.

Although we have focussed on the simply-typed λ-calculus, the techniques presented in this paper can easily be applied to a richer language like PCF containing base-type constants, conditionals and recursion.The game-semantic model we have used in this paper is actually a model for PCF [AJM94,HO94]. Therefore the specification of the analysis itself need not be changed to accommodate PCF.

The proof-net representation of λ-terms can also be extended to include PCF constructs [Mac94]; in particular we can still obtain a isomorphism between the graph and proof-net representations of PCF terms. The proof-net representation of PCF terms will include special nodes for constants, a special form of axiom-link for conditionals and a "looping" cut-link to represent recursion. From the point of view of the analysis algorithm, these graphs are exactly the same as the graphs for simply typed λ-terms, and only a minimal change is required in the algorithm to deal with PCF terms.

There is no necessity to stop at PCF. The game semantic models of programming languages often exhibit very elegant *factorisation results*, which allow language features to be added in a modular fashion. We expect to be able to use the semantic machinery of games to extend the analysis to languages with additional features like side-effects, continuations, non-determinism, etc. In fact the specification of the analysis is expected to be very similar to the ones in this paper; the challenging issue is obtaining representations analogous to proof-nets for developing efficient algorithms.

References

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Proceedings of LICS'98*. IEEE Press, 1998.

[AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In *Proceedings of TACS'94*, volume 789 of *Lecture Notes in Computer Science*, pages 1 – 15. Springer-Verlag, 1994.

[AM97] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. In Peter W O'Hearn and R D Tennent, editors, *Algol-like Languages*, volume 2. Birkhaüser, 1997.

[BKKS87] Henk Barendregt, J R Kennaway, J W Klop, and M R Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 74:191–231, 1987.

[Dan90] Vincent Danos. *Une Application de la Logique Linéaire à l'Étude des Processus de Normalisation (Principalement du Lambda-calcul)*. PhD thesis, Université Paris VII, 1990.

[Deu92] Alain Deutsch. *Modeles Operationnels de Langage de Programmation et Representations de Relations Sur des Langages Rationnels avec Application a la Determination Statique de Proprietes de Partages Dynamiques de Donnees*. PhD thesis, Universite Paris VI, 1992.

[DHR96] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics and abstract machines. In *Proceedings of LICS'96*. IEEE Press, 1996.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1 – 102, 1987.

[Har99] Russell Harmer. *Games and full abstraction for non-deterministic languages*. PhD thesis, University of London, 1999.

[HM97a] Nevin Heintze and David McAllester. Linear time subtransitive control flow analysis. In *Proceedings of PLDI'97*, pages 261–272. ACM SIGPLAN, June 1997.

[HM97b] Nevin Heintze and David McAllester. On the cubic-bottleneck of subtyping and flow analysis. In *Proceedings of LICS'97*. IEEE Press, 1997.

[HO94] J. Martin E. Hyland and C.-H L. Ong. On full abstraction for PCF: Parts I, II and III. Available from [?], 1994.

[Jon81] Neil D Jones. Flow analysis of lambda expressions. In S Even and O Kariv, editors, *Proceedings of ICALP'81*, number 115 in Lecture Notes in Computer Science, pages 114–128, Acre (Akko), Israel, July 1981. Springer-Verlag.

[Lam94] François Lamarche. Proof nets for intuitionistic linear logic I: Essential nets. Available from [?], 1994.

[Lam96] François Lamarche. From proof nets to games: extended abstract. *ENTCS*, 1996.

[Mac94] Ian Mackie. *The Geometry of Implementation.* PhD thesis, University of London, 1994.

[McC98] Guy McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types.* Distinguished dissertations. Springer-Verlag, 1998.

[MH98a] Pasquale Malacaria and Chris Hankin. Generalised flowcharts and games: Extended abstract. In Kim G Larsen, Sven Skyum, and Glynn Winskel, editors, *25th International Colloquium, ICALP'98*, Aalborg, Denmark, July 1998.

[MH98b] Pasquale Malacaria and Chris Hankin. A new approach to control flow analysis. In Koskimies Kai, editor, *Proceedings of CC'98*, volume 1383 of *Lecture Notes in Computer Science*, Lisbon, Portugal, 1998. Springer.

[MH99] Pasquale Malacaria and Chris Hankin. Non-deterministic games and program analysis: An application to security. In *Proceedings of LICS'99*. IEEE Press, 1999.

[Mil89] Robin Milner. *Communication and Concurrency.* International series in computer science. Prentice Hall, 1989.

[Mos97a] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, University of Copenhagen, 1997.

[Mos97b] Christian Mossin. Higher-order value flow graphs. In *Proceedings of 9th International Symposium on Programming Languages, Implementation, Logics, and Programming (PLILP'97)*, 1997.

[MP92] Gianfranco Mascari and Marco Pedicini. Head linear reduction and pure proof net extraction. In *Theoretical Computer Science*, volume 135, pages 111 – 137, 1992.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of POPL'97*. ACM Press, 1997.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[Pal94] Jens Palsberg. Global program analysis in constraint form. In Sophie Tison, editor, *Proceedings CAAP'94*, volume 787 of *Lecture Notes in Computer Science*, pages 276–290. Springer-Verlag, 1994.

[Sam98] Prahladavaradan Sampath. On abstract machines and games. In Valeria de Paiva and Eike Ritter, editors, *Proceedings of the ESSLLI'98 Workshop on Logical Abstract Machines*, Technical Report CSR-98-8. University of Birmingham, 1998.

[Sam99] Prahladavaradan Sampath. *Program Analysis using Game Semantics.* PhD thesis, University of London, 1999.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1991.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages (or) Taming Lambda.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[Wad71] C Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, Oxford University, 1971.