Non-deterministic Games and Program Analysis: An application to security (Extended Abstract)

Pasquale Malacaria and Chris Hankin
Dept. of Computing
Imperial College
LONDON SW7 2BZ
pm5,clh@doc.ic.ac.uk

Abstract

We present a unifying framework for using game semantics as a basis for program analysis. Also, we present a case study of the techniques. The unifying framework presents games-based program analysis as an abstract interpretation of an appropriate games category in the category of non-deterministic games. The case study concerns an application to security.

Keywords: *game semantics, security, linear logic, type systems.*

1 Introduction

Our aim in this paper is twofold: first, we present a unifying framework for previous work on using game semantics as a basis for program analysis; second, we present a case study of the techniques. The unifying framework presents games-based program analysis as an abstract interpretation of an appropriate games category in the category of non-deterministic games [6]. The case study concerns an application to security.

Previous work on games-based program analysis [16, 17] was based on a graphical representation of terms in normal form with interaction links (which we called dotted arrows) to capture the dynamics of computation; this work was restricted to typed, call-by-name languages. Here we show how these graphs can be described by a lax functor between the appropriate category of games and the category of non-deterministic games in [6]. The dotted arrows are recovered by extending this functor to operate on parallel composition. Within this underlying framework we can also model call-by-value languages [5], untyped languages [15] and languages with control features such as exceptions [14].

In the absence of higher order features, the basic notion of security is that, once program variables are given a security clearance, no user able to read values stored in variables with clearance c can gain any information about values stored in variables with a clearance higher than c. Many approaches which have clear correctness results are based on a type system security checker [25, 12] in which the security status of a program component is treated in a very similar way to type information in a typed language. In this setting a correctness result amounts to a soundness proof for the type system. However the analogy between types and security is not completely convincing. A variable of type integer is expected to hold an integer value independently from the context it is in. This is to ensure the validity of operations over this variable and for memory allocation. However security is a much more contextual notion. Consider for example the following program:

$$P = x := y; y := z$$

where x is “low security” variable and z a“high security” variable. In [25] it is not possible to certify the security of P, because y should be (because of z) a“high security” variable and then we would breach security with $x := y$ or y should be (because of x) a “low security” variable and then we would breach security with $y := z$. However it is rather obvious that if y hasn't been declared “high security” before the statement $x := y$ then there is no breach of security in P; the variable y has just become “high security” because of $y := z$ and whatever it has been in the past doesn't matter.

Higher order features introduce additional difficulties. In this setting one can get information about secure data without using assignments. Consider the following example:

```
let lookup x l = if l=[] then false
                 else if x= hd(l) then true
                       else lookup x (tail l)
```

Here whoever uses this function can get information about the elements of the list. The natural requirement for a function f is hence that the security of the application of f to its arguments should be at least as high as the highest security of its arguments. So in the above example even if everybody could be authorised to use `lookup`, the result of `lookup a L` couldn't be read by someone not authorised to read `L`.

This approach to security is the one followed by [12] (and is based on similar ideas used by [22] in the framework of integrity). Both these approaches are however based on types and a rigidity similar to the one we have already noticed in the imperative case arises.

We believe hence that a more reasonable treatment of security should be in terms of the information flow of the program. Unlike types, information flow takes into account temporality so to make possible distinctions like the ones we were discussing above.

The game based approach to security has the following features:

- It is a particular application of a general framework for the static analysis of programs. As a consequence it inherits many properties of this general framework:
  - Correctness of the security analysis based on games is a corollary of a much more general result about the relation of abstract games and operational semantics.
  - The game approach works for a broad spectrum of programming languages with functional, imperative and class-oriented features.
- On the other hand, concurrency is at the moment yet outside the game theoretical approach. However there seems not to be a general agreement concerning what is an information flow for a concurrent language (see the discussion in [25])

1.1 Related work

There have been a variety of frameworks for program analysis proposed in the literature [20]. Recently we have shown how Game Semantics [3] provide a suitable semantic basis for program analysis. Game Semantics are a form of denotational semantics but games contain a lot of intensional information. In [16] we developed a closure analysis for a call-by-name functional language. The analysis algorithm, derived directly from the game semantics, was cubic and essentially correct by construction. In [17] we developed a generalised notion of flow-graph, again based on an abstraction of game semantics, and showed how it could be used in the analysis of class-based object-oriented languages with single inheritance. All of this previous work applies to call-by-name languages.

Banâtre *et al* [8] present a program logic which determines information flows for a language based on Dijkstra's guarded command language.

Denning [10] introduced a lattice model for secure information flow. In essence, variables are assigned security clearances. Secure flow analysis aims to ensure that there are no information flows from higher security variables to lower security variables; the other direction is acceptable. Mizuno and Oldehoeft [18] present an information flow algorithm for distributed object-oriented programs which builds on Denning's work. Mizuno and Schmidt use abstract interpretation techniques to show the correctness of the Mizuno and Oldehoeft flow analysis [19]. Sabelfeld and Sands have recently shown how PERs may be used to model this kind of analysis; they have also extended the work to consider probabilistic information flows [24]. Volpano *et al*[25] present a secure flow analysis based on types; they criticise [19] because the correctness of the analysis is demonstrated with respect to a complex instrumented semantics which is not proven correct. Finally, Heintze and Riecke [12] present the SLam calculus – a lambda calculus extended with security and integrity annotations. Their approach is based on types. Their work is one of the first published works which extends to higher-order programs.

Another analysis that is closely related to secure flow analysis is trust analysis [22]. The aim of secure flow analysis is to ensure that "secure" systems do not leak information. The aim of trust analysis is to ensure that trusted systems to not receive untrusted data.

2 Games

We will be concerned with two player games [3, 13, 6]; we designate the two players by *P*, for Player, and *O*, for Opponent. The player represents the program and the opponent represents the environment. 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 move) and a set of valid positions.

The set of valid positions of a game of type *A* is a prefix closed subset of the set of sequences of moves. In addition, in elements of P_A moves alternate between player and opponent.

Games correspond to types [3, 13] or logical formulas [6] ; 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 alternating sequences of moves 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*. Similarly the tensor game $A \otimes B$ is defined by taking as moves the union of the moves from the two component games, the labelling function is the copairing of the two labelling functions for A and B and the valid positions are alternating sequences of moves constrained such that if we project over either *A* or *B* we get a valid position in the respective game and only opponent is allowed to switch from *A* to *B*.

Given a position *s*, we write s_A to represent the subsequence of *s* with moves in M_A.

To model programs (of a particular type) or proofs of a formula we introduce the notion of *strategy* – a non-empty, prefix-closed set of valid positions.

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 [3] for details). The game $A^{\perp}$ is the same as the game *A* except that player moves become opponent moves and vice versa.

In the category of games, the games are objects and strategies are the morphisms.

Application of one program to another is modelled by parallel composition of the corresponding strategies, followed by hiding the interaction [6]:

$$\begin{array}{lll} \sigma;\tau & = & \{s \upharpoonright A, C \mid s \in \sigma \| \tau\} \\ \text{where} & & \\ \sigma : A \to B, \tau : B \to C & & \\ \sigma \| \tau & = & \{s \in (M_A + M_B + M_C)^* \mid s_1 \in \sigma, s_2 \in \tau\} \\ & & \\ \text{where} & & \\ s_1 & = & s \upharpoonright A, B \\ s_2 & = & s \upharpoonright B, C \end{array}$$

3 Nondeterministic Games

A simple formalisation of this general notion of games has been proposed by [6] in order to model Classical Linear Logic; this implies the ability to model “orthogonality”, i.e. linear negation of games, which amounts to swapping the role of Opponent and Player. As described above strategies are just a prefix closed set of positions, so no contraint of “deterministic behaviour” is required; the tricky point of this general notion of games is to show that these nondeterministic strategies do indeed compose and give rise to a category $\mathcal{C}$.

For our purposes we are interested in $\mathcal{A}_I$, the multiplicative intuitionistic subcategory of $\mathcal{C}$, i.e. we consider only $\otimes, \multimap$ game constructors and restrict positions to start with an Opponent move.

We are going now to define a mapping R (R for repetition) on objects of $\mathcal{A}_I$ as follows:

Given a game A, $R(A)$ has the same moves and labeling function as A and a sequence of moves s is a position in $R(A)$ if there exists a labeling ℓ of s such that $\ell(s)$ is a position in $\otimes^\omega A$[1].

Hence $R(A)$ allows to play A *ad libitum*.

The *abstract games* category is $\mathcal{A}$ the subcategory of $\mathcal{A}_I$ whose objects are well opened games (a game is well opened if all of its positions begin with the same unique move).

We interpret normal forms of type $A \to B$ in $\mathcal{A}$ as nondeterministic strategies in $R(A) \multimap B$; this is nothing more than the usual linear logic decomposition of the intuitionistic implication; however by using $R(A)$ we allow the same move to occur several times in a position. This is an important ingredient in our framework because it allows the composition to nondetermistically connect a procedure with all its possible call sites. This possibility is what distinguishes the program analysis of a program from its real evaluation, where the the exact sequence of calls must be determined.

Although the categorical structure of $\mathcal{A}_I$ has an interest in its own, in this paper we are not going to study in detail its properties. For our purposes it is enough here to be able to give a mathematical definition for the abstract interpretation; this require some form of functor (defined in the next section) togheter with some basic properties w.r.t. game constructors. This will be enough in order to give a precise mathematical meaning to the constructions we introdued in earlier works [16, 17] and (given the abstraction of the functor) to extend them to other game models.

Nondeterministic games form a Cartesian Closed Category, hence provide a model for functional languages; however it is a rather loose model where a game interpreting a type has far too many strategies which are not definable in the language. Indeed one of the achievements of game semantics has been to provide a solution to the full abstraction problem for several languages, the first being PCF [3, 13]. These results have been possible by defining more restrictive categories of games than the one we have seen so far.

The game model we are mainly interested here is the fully abstract model for PCF (for technical convenience we will use the Hyland-Ong, Nikau model [13, 21]); We will denote by $\mathcal{C}_{cbn}$ the subcategory of [13] with objects well opened games where the intensionally fully abstract model lives.

More recent full abstraction results (for Idealised Algol[5], Idealised Algol with general references [2], PCF extended with control [14]) have been obtained by carefully relaxing some of the conditions of innocence or bracketing. For example knowing strategies (i.e. strategies that are not innocent) which are well-bracketed are sufficient

[1] Here $\otimes^\omega A$ is the unbounded tensor game which has as moves the disjoint union of omega copies of A, labelling in each copy is the same as A and positions are Opponent starting, alternating sequences of moves where only Opponent can switch component and such that the restriction to each component is a play in A.

to model call-by-name functional languages with first-order references.

4 The Abs operation

We are now going to define a map $\texttt{Abs}$ from the objects of $\mathcal{C}_{cbn}$ to the objects of $\mathcal{A}_I$. First define an equivalence operation on answer moves in a game A as follows: $a \simeq a'$ iff both a and a' are justified by the same question. Given a game A, $\texttt{Abs}(A)$ is then the game whose moves are the question moves of A stripped of their justification pointers and equivalence classes of answer moves of A stripped of their justification pointers, $\lambda_{\texttt{Abs}(A)} = \lambda_A$, and a position in $\texttt{Abs}(A)$ is a sequence of moves which is obtained from a position in A by stripping it of its justification pointers and replacing answer moves with their equivalence classes.

Notice the following property of $\texttt{Abs}$:

Lemma 1

- $\texttt{Abs}(A \otimes B) \subseteq \texttt{Abs}(A) \otimes \texttt{Abs}(B)$
- $\texttt{Abs}(A \multimap B) \subseteq \texttt{Abs}(A) \multimap \texttt{Abs}(B)$
- $\texttt{Abs}(A \times B) = \texttt{Abs}(A) \times \texttt{Abs}(B)$
- $\texttt{Abs}(!A) = R(\texttt{Abs}(A))$

The reason for these inequalities (instead of equations) is not trivial: It depends on the visibility condition. Notice also that the for the first inequality we are considering the larger categories of Hyland-Ong and non-deterministic games (because $(A \otimes B)$ is not necessary well-opened).

$\texttt{Abs}$ can be lifted to operate on arrows in the obvious way, i.e. given a stragegy σ, $\texttt{Abs}(\sigma)$ will be the set of positions obtained by stripping each position in σ of its justification pointers and replacing answer moves with their equivalence classes. Hence $\texttt{Abs}(\sigma)$ is a prefix closed set of positions, i.e. a strategy in $\mathcal{A}$. It is easy to see that the following properties hold;

- $\texttt{Abs}(1_A) \subseteq 1_{\texttt{Abs}(A)}$
- $\texttt{Abs}(\sigma;\tau) \subseteq \texttt{Abs}(\sigma);\texttt{Abs}(\tau)$

The map $\texttt{Abs}$ is hence a Lax Functor [7]; the fact that $\texttt{Abs}(\sigma;\tau) \subseteq \texttt{Abs}(\sigma);\texttt{Abs}(\tau)$ means that the abstract interpretation of a strategy (i.e. of the semantics of a program) is a safe approximation of its semantics.

We will define *abstract strategies* to be the image under $\texttt{Abs}$ of strategies representing programs. Notice how by this means strategies interpreting programs are finitized; e.g. the identity on integers $\{qqnn \mid n \in \omega\}$ becomes $\{qq**\}$. Fixed points can be represented synthetically by using cyclic graphs.

Hence the $\texttt{Abs}$ image of the call-by-name model in $\mathcal{A}$ provides a finitary representation for normal forms[2] whose properties and complexity we have studied in [16, 17]. In order to perform program analysis we still need to finitely represent the interaction between normal forms. Remember that in all categories of games, the composition is defined by parallel composition plus hiding [3], i.e. given $\sigma : A \to B$ and $\tau : B \to C$, first define a set of sequences of moves $\sigma||\tau$ where each sequence is a possible interaction between σ and τ and then remove from these sequences all moves played in the shared subgame B.

We hence define the *abstract interaction* between σ and τ as $\texttt{Abs}(\sigma)||\texttt{Abs}(\tau)$. The laxness of $\texttt{Abs}$ is actually a consequence of the following inequality:

$$(*) \quad \texttt{Abs}(\sigma||\tau) \subseteq \texttt{Abs}(\sigma)||\texttt{Abs}(\tau)$$

which in turn states that the abstract interaction between σ and τ is a safe approximation of their evaluation.

4.1 Safeness

We are now going to relate the abstract strategies and abstract interaction with the operational semantics of the language. To keep things as simple as possible we will restrict ourselves to PCF terms with no constants which are in $\eta-$long form and in *normalised form* (i.e. of the shape $MN_1 \dots N_p$ with all M, N_i in normal form).

First define the *Linear Head Reduction* (LHR) [9] of a lambda term M as follows: at each step consider the left-most 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.

Given a normalised term $MN_1 \dots N_p$ we have the following property:

Lemma 2 *The LHR of a normalised term $MN_1 \dots N_p$ to weak head normal form generates a unique sequence of pairs $(x_1, S_1), \dots, (x_n, S_n)$ where for j even (resp. j odd) x_j is a variable in N_k (for some k)(resp. a variable in M) and S_j a subterm in M (resp. in N_k (for some k)). Call such a sequence the LHR sequence of $MN_1 \dots N_p$.*

It is well known that given a PCF strategy σ corresponding to a normal form S we can identify player questions in

[2] The infinitary nature of the interpretation of normal forms in game semantics comes from two reasons: On one side a strategy can be infinite because a question has an infinite number of answers – this infinity is eliminated by identifying all answers with $*$. On the other side infinity can be caused by having positions of unbounded length – this is what happens in the case of the fixed point constant; however this infinity can be expressed as a regular expression, hence a finite graph.

σ as variables in S and Opponent questions in σ as subterms of S; this is the essence of the decomposition lemma. Given a sequence of question moves s let $\lambda(s)$ the sequence of corresponding variables/subterms.

Given $\mu, \nu_1, \ldots, \nu_p$ strategies interpreting $M, N_1, \ldots, N_p$ we have then (assuming for simplicity that $MN_1 \ldots N_p$ is of basic type):

Proposition 1 *Let s be the play generated by $\mu\|\nu_1\| \ldots \|\nu_p$, s_1 the subsequence of s in the hidden part (i.e. s_1 is not in $\mu; \nu_1; \ldots; \nu_p$) and containing only question moves. Then $\lambda(s_1) = x_1, S_1, \ldots, x_n, S_n$ where $(x_1, S_1), \ldots, (x_n, S_n)$ is the LHR sequence of $MN_1 \ldots N_p$.*

This proposition is basically proved in [9]. Its meaning is that the LHR operational semantics corresponds to the parallel composition in game semantics.

As a consequence of the above proposition and inequality $(*)$ we have that the LHR sequence of $M, N_1, \ldots, N_p$ is included in the abstract interaction between $\mu, \nu_1, \ldots, \nu_p$ and hence the abstract interaction is safe.

4.2 Charts

Notice that an abstract strategy μ, $\texttt{Abs}(\mu)$ is just a finite graph whose nodes are the moves in $\texttt{Abs}(\mu)$ and there is an edge between m and m' if $smm's' \in \texttt{Abs}(\mu)$ for some sequence s, s'; we will call these *solid edges*. Similarly the abstract interaction $\texttt{Abs}(\mu)||\texttt{Abs}(\nu)$ will give rise to edges between $\texttt{Abs}(\mu)$ and $\texttt{Abs}(\nu)$ which we will call *dotted edges*.

We define hence the *chart* of a program $M, N_1, \ldots, N_p$ as the graph obtained first by generating the graphs $\texttt{Abs}(\mu), \texttt{Abs}(\nu_1), \ldots, \texttt{Abs}(\nu_p)$ and then connecting these by the dotted edges generated by the abstract interaction.

Notice that each node in the graph will be either a question or an answer (because nodes are moves) and that each dotted edge will be either a *question* edge (if it connects two questions) or an *answer* edge (if it connects two answers).

5 Generality of the game approach

An important question is how general is our approach; the following result shows that we can build charts for most of the existing game models.

Proposition 2 *The lax functor* Abs *is well defined over the game models for the following languages:*

- *Call-by-Name PCF [3] and Idealised Algol [5].*
- *Call-by-Value PCF and Idealised Algol [4].*
- *Untyped Lambda calculus [15].*
- *Functional languages with control operators [14].*

This result depends on the fact that these models bear a strong similarity to the call-by-name PCF model; for example in [4] (section 3.2) it is shown that a fully abstract game model for a call-by-value Idealised Algol can be obtained in a subcategory of the category in which the fully abstract game model for Idealised Algol is constructed. The key ingredient in the fully abstract model for functional languages with control is to relax the bracketing condition; since the map Abs doesn't take into account the bracketing condition its relaxation is not influential to the construction.

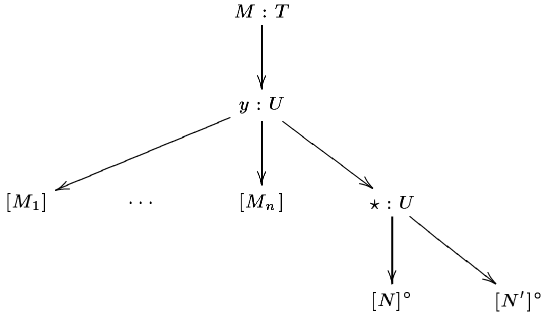
There are however game models like the one for general references [2] for which the Abs map is not well defined; for example $\texttt{Abs}(A \multimap B) \not\subseteq \texttt{Abs}(A) \multimap \texttt{Abs}(B)$. To find an abstract interpretation for this model is a matter for future investigation.

6 Constructive definition of Charts.

In [17] we give a more constructive definition of this abstract interpretation for call-by-name Idealised Algol. We give a syntax-based translation from each normal form M to an abstract strategiy (represented by a graph) noted $[M]$.

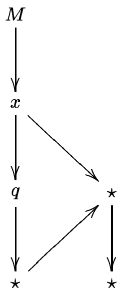
This step corresponds to the image under Abs of normal forms. To see why this is consider the key case: Take a normal form $M = \lambda x_1 \ldots x_n.\texttt{cond}(yM_1 \ldots M_p)NN'$. The strategy associated to this normal form will, on receiving the initial Opponent question, ask for the variable y; at this point either Opponent answer it or is going to play some of the strategies associated to some M_i; eventually Opponent will answer the y question; Player will use that answer a_y to play N (if a_y is true) or N' (if a_y is false); however the initial question of N and N' is not played and their final answer is considered as the answer to the initial Opponent question for the whole term.

The abstract strategy corresponding to this complex dynamics can be expressed by the following tree [17]:

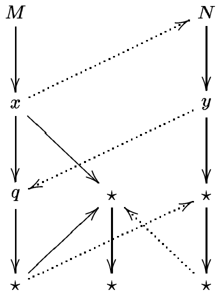


where T and U are the types of the term M and the variable y and $[H]^\circ$ is the graph $[H]$ pruned of its root.

This tree is an optimization of $\texttt{Abs}(\lambda x_1 \ldots x_n.\texttt{cond}(yM_1 \ldots M_p)NN')$ where we would have more edges; the paths generated by these additional edges can be recovered by the abstract interaction and are hence eliminated in the optimized abstract strategies. For example consider $M = \lambda x.\mathtt{cond}(x\mathtt{tt})\ \mathtt{ff}\ \mathtt{tt}$; $\mathtt{Abs}(M)$ is the following tree:



the diagonal edge between the two $\star$ nodes is absent in the optimized version; the paths added by that edge are however recovered in the abstract interaction, e.g. if we consider the abstract interaction between M and $N = \lambda y.y$ we get:



In the constructive definition of charts the dotted arrows are built using a syntactic algorithm which uses type and subterm information.

An important point to notice is that the complexity of the abstract interpretation of a family of strategies is linear and the algorithm to add abstract interaction is quadratic.

Using this abstract representation we have so far devised algorithms in the following fields of program analysis:

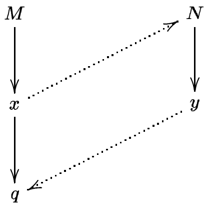
- Generalised flowcharts as a basis for higher order dataflow analysis [17].
- Closure analysis [16].

In the paper [17] we show how a generalised (higher-order) flowchart can be constructed by considering certain paths through these graphs. The use of the term "flowchart" is justified by the fact that the usual flowcharts are the particular case of the general construction in the case when the program doesn't contain any higher-order component.

To define these generalised flowcharts one starts by identifying nodes in the graphs that correspond to subterms; notice that commands are just particular subterms of the whole program. In general this can be done by using the decomposition lemma, a key technical lemma in the proof of full abstraction for game models. Once this has been done a preorder $<$ between these nodes is defined based on a notion of "reachability" by saying that $s < s'$ if s' is reachable from s. The guiding idea is that $s < s'$ if there is a possible evaluation where s happens before s'.

The worst case complexity for building a generalised flowchart is cubic in the size of the program; however this situation is rarely encountered; moreover for restricted languages generalised flowcharts are easier to build; for example the algorithm restricted to imperative programs has linear complexity.

The closure analysis of [16] involves a further abstraction: we forget answer moves. The closure analysis for a higher order program aims to determine, given a program point p, which are all closure p can jump to; the key case to consider is what are all possible substitutions of a bound variable with an abstraction. This computation can be done by some kind of (dotted/solid) alternating path in the chart and doesn't involve visiting question nodes or edges; for example to determine closure analysis for the program MN with $M = \lambda x.\mathtt{cond}(x(\lambda z.z)))\ \mathtt{ff}\ \mathtt{tt}, N = \lambda y.y\mathtt{tt}$ the following graph is sufficient:



This graph allow us to determine that the variable x will be substituted with N and the variable y will be substituted with q (which corresponds to $\lambda z.z$). The closure analysis for the subterm collects all closures that are accessible in this way. It is shown [16] that this analysis is as accurate as state of the art algorithms and has the same complexity (cubic in the size of the term). The big advantage is that the analysis is constructed directly from the semantics and is correct-by-construction.

7 Information Flow

We now consider a case study of the technique: the use of an information flow analysis to check security for Idealised Algol.

For first-order languages, the basic notion of security is that each variable has an associated security level and no user is able to read variables at a higher level than the user's clearance. For example, a variable on the left hand side of an assignment is the user of any variable appearing in the right hand side. Such dependencies can be captured via the notion of *information flow*.

In this section we define a generalisation of information flow for Idealised Algol. The algorithm here described performs an interprocedural analysis (i.e. first order Idealised Algol program analysis) to detect security breach. The idea is to take the chart of a program and define some kind of paths; these paths are then used to build another graph (the flow graph) for the program; security is then computed in terms of connectivity in this flow graph.

The next section extend this algorithm to security for higher order programs by adding additional nodes to the flow graph; security is again computed in terms of connectivity in this flow graph; however an additional algorithm has to be used (the control-flow algorithm described in [16]) to compute all possible abstractions which can be substituted for a bound variable.

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

In order to define a particular class of paths over which information flow analysis is based we need the following lemma:

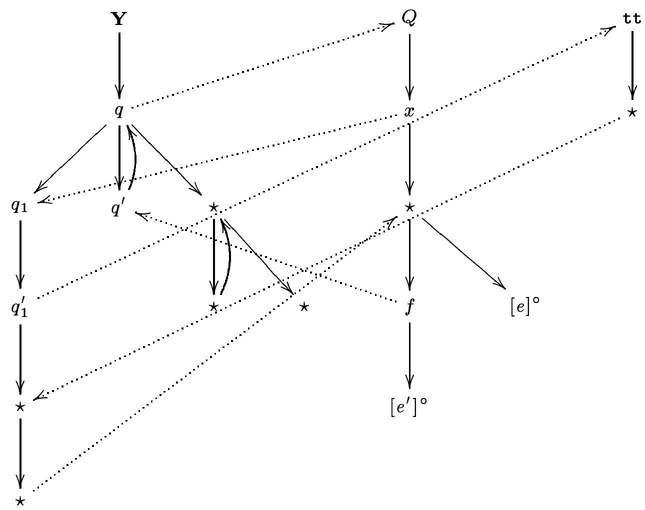
Lemma 3 *Let $M_1 \dots M_p$ a term in normalised form and n be a node in $[M_1]$ (resp. in $[M_i]$, $1 < i \leq p$) with m outgoing dotted edges; then all these edges have as target exactly one $[M_i]$, $1 < i \leq p$ (resp. $[M_1]$) and one of these m edges is above the others (in the tree sense); we call this the* upper edge *of n.*

Define a *standard 1-valid path* (SVP) as the following valid path: Start from the root of the graph and do an alternating path over unvisited nodes; if you are at a node n with more than one (unvisited) outgoing edge then choose the upper edge of n. If you are at a (non answer) node and you have visited all its outgoing edges then jump to (one of) its answer(s).

Given nodes a, b we say that b is *1-reachable* from a (written $a <_1 b$) if there is a 1-valid path in which a occurs before b.

The number of SVPs is a function of the number of tests of the program and corresponds to a possible execution trace.

Here is an example of a chart $\mathbf{Y}Q\mathtt{tt}$ with $\mathbf{Y}$ the fixpoint constant , $\mathtt{tt}$ the constant true and $Q = \lambda f x.\mathtt{cond}\ xe(f(e'))$ with e, e' unspecified expressions:



An SVP here would be e.g. $\mathbf{Y}, q, Q, x, q_1, q_1', \mathtt{tt}, \star, \star, \star, \star, f, q', q$. This SVP would tell us that the Q is a recursive function with recursion parameter f (because the closure associated to f is Q, f is 1-reachable from q, q is 1-reachable from f and the value of q is Q).

Notice how this kind of informations could be equally computed with the valid path $\mathbf{Y}, q, Q, x, \star, f, q', q$ (i.e. we avoid the argument $\mathtt{tt}$), hence "virtual" SVP can extract informations about evaluation in a modular way.

Given the family of SVPs of a program P we are going to define the *information flow* graph of P. The nodes of this graph are occurrences of variables and the edges are information *flows*. The occurrences of variables are associated with program points in the flowchart and the flows are generated as follows:

We will assume that each time we build a new flow from x to y, we also create a flow from the last occurrence of x we met as target of a flow to this new occurrence of x. Following a SVP s and meeting a node n, we reason by cases:

- n is an assignment $x := E(Y)$ (where $Y = y_1, \dots, y_n$) then create a flow from each node corresponding to these occurrences of y_i to x.
- n is a conditional: Then n could correspond to an "if...then...else" or to a "while" statement.
  - In the first case create a flow from the variables Y in the guard to all variables appearing in the branches.
  - In the second case create a flow from the variables Y in the guard to all variables following in the program. Moreover starting from the bottom of the body of the `while` going up to its guard, for all x, create a flow from the last occurrence

of x we met as target of a flow to the previous occurrence of x.

- n is a bound variable of type $Var[X]$. In that case follow s until a node m corresponding to a variable of type $Var[X]$ and create a flow from m to n.

Using information flow how are we going to compute security and integrity? First some tags should be added to the structure, in the case of security one should add the adequate security tags to the nodes corresponding to the variables with that security class. We say that there is no security breach at the node n if the security associated to all nodes from which there is a path of flows to n isn't higher than the security associated to n. Integrity works in the same way by using different "trust" tags.

A more extensional presentation of this algorithm can be given and we now proceed to do this.

Given a flowchart for a program P and a program point p in P the *guards set* at p (noted by G_p) is defined as the set of variables in `cond` or `while` guards embedding p or `while` guards preceding p.

We are now going to define by mutual recursion two sets IF_p and $\mathcal{I}$ which are recursively computed going backwards from a program point p. In the definition of these sets we use an auxiliary function *store*. We write $x \sqsubseteq y$, where y is of type *Var[X]*, to mean that $x <_1 n$ where n is the *cell* node in the strategy for y and that there are no intervening cell nodes. The function *store* is defined as follows:

$$store(x) \quad = \quad \{y \mid x \sqsubseteq y\}$$

We lift *store* to sets of variables in the obvious way.

The *indirect flow* to p (noted as IF_p) is defined by

$$IF_p = G_p \cup \bigcup_{x \in store(G_p)} \mathcal{I}(x := E(Y))$$

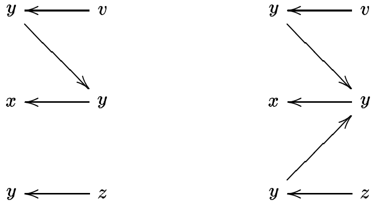
Notice that computing backwards means that the $x := E(Y)$ are the first assignments to x (for $x \in store(G_p)$) met going backwards in the flowchart starting from the program point p.

The *information flow* to a variable x in a statement $x := E(Y)$ at a program point p (where $Y = y_1, \ldots, y_n$) is defined as the following function (which again is recursively computed going backwards from the program point p):

$$\mathcal{I}(x := E(Y)) = Y \cup IF_p \cup \bigcup_{z_i \in store(y_i)} \mathcal{I}(z_i := E(Z))$$

Example: Given the program $P \equiv x := y; y := z$ we have $IF_{y:=z} = \emptyset$ and $\mathcal{I}(y := z) = \{z\}$. For $P' \equiv$ `cond` $a\ (x := y; y := z)$ we would have $IF_{y:=z} = \{a\}$ and $\mathcal{I}(x := y) = \{y\}$. However for $P' \equiv$ `while` a `do` $x := y; y := z$ `od` we would have $\mathcal{I}(x := y) = \{y, z\}$

Here are the flow graphs for $y := v; x := y; y := z$ (left) and $y := v;$ `while tt do` $x := y; y := z$ `od` (right).



7.1 Correctness

The notion of correctness we are going to use is the same as in [8], i.e. if y is not in the information flow of a variable x then a computation starting with any state which differs only for the values given to y will produce (at the program point p) the same value for x. So there is no communication between x and y.

More formally, given two states τ, τ' define the relation $\tau \mathcal{A}_p \tau'$ if τ and τ' agree on all variables in $\mathcal{I}_p$.

We have then the following

(here $< P, \tau >$ is a configuration in the operational semantics of the program)

Proposition 3 *$\tau \mathcal{A}_{x:=E(Y)} \tau'$ and $< P, \tau > \rightarrow^* < x := E(Y), \sigma >$, $< P, \tau' > \rightarrow^* < x := E(Y), \sigma' >$ implies that $\sigma \mathcal{A}_{x:=E(Y)} \sigma'$*

Notice that this proposition implies that $x := E(Y)$ has the same semantics in σ and in σ' because $\sigma \mathcal{A}_{x:=E(Y)} \sigma'$ and thus well formalizes the intuition of correctness.

8 Higher-order Security

So far, we have concentrated on breaches of security that result from accesses to storage locations. For first-order programs, this may be sufficient. The situation changes when we consider higher-order programs; in this setting it is important to associate security with functional parameters (which essentially behave as mobile code) and parameters of other types (for example lists).

In order to illustrate our approach, we consider the following program:

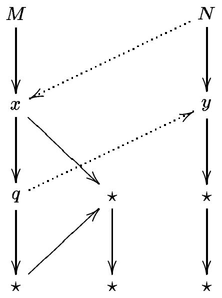
```
let  fold =    λf l b. new l'.
               acc := b; l' := l;
               while l' <> null do
                 (acc:= f acc (hd l');
                 l':=(tl l'));

     map =     λf l. if l = null
               then null
               else
               (f (hd l)):(map f (tl l));

     lookup =  λx y. x = y;
in
     new result acc.
     fold or (map (lookup 10) l) false;
     result := acc;
```

The program defines an imperative version of “fold” which combines the elements of a list using a binary operation, a functional version of “map” which applies a function to every element of a list and a “lookup” function which tests for the equality of two values. The main body of the program tests some list, `l`, to determine if 10 is an element of it. The answer (a boolean) is assigned to `result`.

The algorithm for building flows must be modified to account for bound variables of higher type. First the nodes of the flow graph include now the nodes of the chart. Supposing the bound variable occurs at node *n* then we must follow each SVP, *s*, from *n* until we encounter a node *m* corresponding to a value (a constant or a closure) of the appropriate type and build a flow from *m* to *n*. In following *s*, we may encounter a node representing an application $xt_1 \ldots t_n$; in this case, we follow the path from the head variable using the “cut” operation of [16] to remove binders corresponding to the arguments.

Security (or integrity) is then computed by extending the tagging to subterms; a “no breach” of security is computed as in the previous section by adding the following condition: If an expression E has one of its bound variables target of a flow, then the security clearance of E has to be at least the same as the one of the source of the flow.

Let’s for example consider again MN with $M = \lambda x.\texttt{cond}(x\texttt{tt})$ `ff tt` and $N = \lambda y.y$; applying the algorithm we get the following flows (noted by dotted arrows; notice their inverted direction w.r.t the dotted arrows in the original chart):



Hence if we suppose that the constant `tt` in M has security clearance s then N should also have security at least s; henceforth x should have at least the same security and eventually M would have to have at least security s as well.

Notice that in an expression like $(\lambda xy.y)a$ the security of the whole expression is independent of the security of a (there is no flow from a to x); this shows how our approach is more flexible than a type oriented one.

Then correctness in this higher-order setting can be stated as follows:

Proposition 4 *Suppose k is a constant or a closure. If k is substituted for a variable x during the evaluation of a program then, if the algorithm fails to detect a breach of security,* $\varsigma(k) \leq \varsigma(x)$*, where* $\varsigma(v)$ *returns the security clearance of* v*.*

The proof of this proposition follows from the correctness of the closure analysis algorithm ([16]).

Now suppose that the list `l` has some security clearance *c*. The sub-expression `map (lookup 10) l` inherits security *c*. As a consequence, the variable `l'` must have security at least *c* and so too must `acc`. Finally, `result` must also have security at least *c* in order to avoid a breach of security.

8.1 Complexity

The worst case complexity for all of the algorithms outlined is $O(n^3)$, where *n* is the size of the Idealised Algol term. This result follows directly from the underlying complexity of the construction of standard 1-valid paths.

9 Extending Security Analysis

We can exploit the unifying framework to extend the results of the last two sections to call-by-value and untyped languages. The details of the extension are routine but require minor reformulations of some of the technical results (for example Lemma 3).

10 Conclusions

We have shown how the category of games from [6] provides a unifying framework for our previous work on program analysis for call-by-name Algol-like languages and call-by-value languages with various extensions. We proceeded by showing how the category for call-by-name languages could be abstracted in the underlying category via a lax functor. We have also shown how the abstraction of the category for call-by-value languages forms a subcategory of the image of call-by-name one. The first main contribution of this paper has been the identification of the category of [6] as a unifying framework and the development of the `Abs` lax functor. The second main contribution has been to show how the information flow-based approach to security can be generalised to higher-order languages based on our notion of generalised flowchart.

Our eventual target is to be able to analyse languages such as Java. The ability to have references to objects is an essential step in this enterprise; the work on games with general references is a basis for this. Games with control allow us to model exception mechanisms.

A major remaining obstacle is our current inability to model concurrent programs. This remains a topic for future work.

Acknowledgements

We are grateful to our colleagues, particularly Samson Abramsky, Guy McCusker and Russ Harmer, from the TCOOL project for their support and encouragement. We thank Vincent Danos for helpful suggestions. Thanks also to the UK Engineering and Physical Sciences Research Council for funding TCOOL and supporting the first author through an Advanced Research Fellowship.

References

[1] M. Abadi, Secrecy by typing in security protocols. In *Proc. TACS*, Springer-Verlag, September 1997, pp 611-638.

[2] S. Abramsky, K. Honda and G. McCusker, A fully abstract game semantics for general references. In *Proc. LICS'98*, IEEE Press, 1998.

[3] S. Abramsky, R. Jagadeesan and P. Malacaria, Full abstraction for PCF (extended abstract). In *Proc. TACS'94*, LNCS 789, pp 1–15, Springer-Verlag, 1994.

[4] S. Abramsky and G. McCusker, Call-by-value games. In *Proc. CSL'97*, LNCS, Springer-Verlag, 1997.

[5] S. Abramsky and G. McCusker, Linearity, sharing and state: a fully abstract game semantics for Idealised Algol with active expressions. In *Algol-like Languages, Volume 2*, P.W. O'Hearn and R. D. Tennent (editors), Birkhaüser, 1997.

[6] P. Baillot, V. Danos, T. Ehrhard and L. Regnier, Believe it or not: AJM's games model is a model of classical Linear Logic. In *Proc. LICS'97*, IEEE Press, 1997.

[7] P. Baillot, V. Danos, T. Ehrhard and L. Regnier, Timeless games. In *Proc. CSL'97*, LNCS, Springer-Verlag, 1997.

[8] J. Banâtre, C. Bryce and D. Le Métayer, Compile-time Detection of Information Flow in Sequential Programs, in *Proc. of the European Symposium on Research in Computer Security*, Springer LNCS 875, 55–73, 1994.

[9] V. Danos, H. Herbelin and L. Regnier, Game Semantics and Abstract Machines. In *Proc. LICS'96*, IEEE Press, 1996.

[10] D. Denning, *A Lattice Model of Secure Information Flow*, Communications of the ACM, 19(5), 236–242, 1976.

[11] P. Freyd, Algebraically Complete Categories. In *Proc. 1990 Category Theory Conference*, LNCS, Springer-Verlag, 1991.

[12] N. Heintze and J. G. Riecke, The SLam Calculus: Programming with Secrecy and Integrity, in *Proc. POPL'98*, ACM Press, 1998.

[13] M. Hyland and L. Ong, On full abstraction for PCF: I, II and III. 130 pages, ftp-able at `theory.doc.ic.ac.uk` in directory `papers/Ong`, 1994

[14] J. Laird, Full abstraction for functional languages with control. In *Proc. LICS'97*, IEEE Press, 1997.

[15] G. McCusker, Games and Definability for FPC, The Bulletin of Symbolic Logic, 3, 1997.

[16] P. Malacaria and C. Hankin, A New Approach to Control Flow Analysis. In *Proc. CC'98*, LNCS 1383, pp 95–108, Springer-Verlag, 1998.

[17] P. Malacaria and C. Hankin, Generalised Flowcharts and Games. In *Proc. ICALP'98*, LNCS 1443, Springer-Verlag, 1998.

[18] M. Mizuno and A. E. Oldehoeft, Information Flow Control in a Distributed Object-Oriented System with Statically Bound Object Variables, in *proc. 10th National Computer Security Conference*, 1987.

[19] M. Mizuno and D. Schmidt, *A Security Flow Control Algorithm and its Denotational Semantics Correctness Proof*, Formal Aspects of Computing, 4(6A), 722–754, 1992.

[20] F. Nielson, H. R. Nielson and C. Hankin, *Principles of Program Analysis: Flows and Effects*. to appear, 1999.

[21] H. Nikau, Hereditary sequential functionals, in *proc. of the symposium on logical fundations of computer science*, Springer LCNS, 1994.

[22] J. Palsberg and P. Ørbæk, Trust in the λ-calculus, in *proc. of the 1995 Static Analysis Symposium*, Springer LNCS 983, 1995.

[23] J. C. Reynolds The essence of Algol. In J. W. de Bakker and J. C. van Vliet (eds), *Algorithmic Languages*, pp 345–372, North-Holland, 1981.

[24] A.Sabelfeld, D.Sands. A per model of secure information flow in sequential programs. In *Proc. Esop 99*, LNCS 1999.

[25] D. Volpano, G. Smith and C. Irvine, *A Sound Type System for Secure Flow Analysis*, Journal of Computer Security, 4(3), 1996.