JAN JÜRJENS

GAMES IN THE SEMANTICS OF PROGRAMMING LANGUAGES – AN ELEMENTARY INTRODUCTION

ABSTRACT. Mathematical models are an important tool in the development of software technology, including programming languages and algorithms. During the last few years, a new class of such models has been developed based on the notion of a mathematical game that is especially well-suited to address the interactions between the components of a system. This paper gives an introduction to these game-semantical models of programming languages, concentrating on motivating the basic intuitions and putting them into context.

1. INTRODUCTION

The importance of reliability in software products is by now well-known (most commonly known example for a source of concern is the "year 2000 bug"). One aim in the design of new programming languages is to try to minimize the occurrences of mistakes through appropriate design (classical example is to encourage programmers to program in a structured way, as achieved through the design of the programming language Pascal by providing programming constructs such as procedures). More suitable programming languages enable a more efficient and more secure development of software.

Providing mathematical models for programming languages is an important step in this direction. Their purpose is to serve as a basis for understanding and reasoning about how programs behave. On the one hand, they can be used for analysis and verification, on the other, there have been significant examples of the design of new programming language principles influenced by the mathematical foundations (for instance the influence of the lambda-calculus on the development of the functional language ML).

For a long time (ca. 1950–1980) these models were functional in nature: execution of a program was thought of as computation of a function (as opposed to an interactive process). Using this model allowed the development of the notions of correctness of a program with respect to its specification. These models could be classified as *operational semantics* respectively *denotational semantics*. Different structures also relating inputs to outputs

Synthese **133:** 131–158, 2002.

in a functional or relational way (namely Turing machines) were employed to model computational complexity.

- Operational semantics employs an evaluation relation $\Downarrow$: $M \Downarrow c$ means that the "program" M converges to the canonical form ("value") c ("big-step"-semantics).
- In denotational semantics one employs mathematical methods to use one's intuition about specific mathematical structures to reason about programming languages. Programs are interpreted compositionally in the structures that traditionally are order-theoretic: A term $\mathsf{op}(M_1, \ldots, M_n)$ consisting of an operation op and subterms M_i is interpreted by composing the interpretations of the operation and of the subterms:

$$[\mathsf{op}(M_1, \ldots, M_n)] := [\mathsf{op}]([M_1], \ldots, [M_n]).$$

Both kinds of semantics have been very successful, but also have disadvantages: Operational semantics is syntax-dependent and thus too explicit for a nice mathematical theory. In denotational semantics, the programs are modeled extensionally (i.e., showing only input/output-dependencies and no aspects of the actual computation process) which abstracts from their dynamics. While this has been an adequate approach to the traditional forms of (functional) computation, the rise of interest in distributed systems (of which the most commonly known is the internet) in recent years has called for a model that takes account of the interactions between components of a system (or equivalently, between a system and its environment). Moreover, this approach also models appropriately the realization of functional computation. More speculatively, an intensional model of computation (i.e., one that reflects some properties of the process of computation) could perhaps also be used to model computation-related aspects like computational complexity.

These observations beginning in 1992 in Abramsky and Jagadeesan (1992) (and independently in Hyland and Ong (1992)) led to the construction of very satisfactory game-semantical models for linear logic (a resource-sensitive logic introduced in Girard (1987); another model had been given in Blass (1992), but with non-associative composition). These model were intensional in nature: thus the usual completeness results, stating that provability of a formula is reflected in the model, were strengthened to "full completeness" results where each proof is itself represented. Another games model for linear logic was given in Lamport (1994), while the ones in Lafont and Streicher (1991) or Mey (1994) (the latter for predicate logic without contractions) are not intensional.

Subsequently, this led in 1993 to the development of intensional game-theoretical models in the semantics of programming languages independently by Abramsky et al. (1994); Hyland and Ong (2000) and Nickau 1996). These models proved to be very useful and provided e.g., a solution for the probably best-known open problem in the semantics of programming languages, the "Full Abstractness Problem" for the programming language PCF (Plotkin 1977), by giving the first syntax-independent fully abstract model. PCF is a higher-order functional programming language that essentially is a fragment of any programming language with higher-order procedures (for instance any expressive enough object-oriented programming language).

Precursors to these game-theoretical models can be seen in Joyal (1977), where for the first time a category of games is defined, and in the work of Kleene on recursive functionals, and of Berry and Curien on sequential algorithms in Berry and Curien (1982).

In another line of research, game-semantical methods have so far had a number of other applications, including in Abramsky and Jagadeesan (1994) an alternative realization of the "Geometry of interaction" program (initiated in Girard (1989) and developed in a series of papers).

In this paper we would like to give an accessible introduction to the games model of PCF while concentrating on motivating the basic intuitions and putting them into context.

2. GAMES – INFORMAL DEVELOPMENT

Game theory was founded in the beginning of the century with works by Zermelo, Borel and von Neumann on parlour games. In the 1950's John Nash made his famous contributions to non-cooperative game theory and to bargaining theory that he later received the Nobel Prize of economics for (together with J. C. Harsanyi and R. Selten).

The use of game-theoretical methods in logic (with the best-known example the Ehrenfeucht-Fraissé games used in (finite) model theory) also originated at the beginning of this century and is still a strong field of research. Similarly game-theoretical methods have been used in models of concurrency/reactive systems (for an introduction into the latter field (cf. Merz 2002), the modeling of interactive protocols, natural language semantics (Hintikka games (Hintikka and Sandu 1997)) etc.

2.1. *Lorenzen Games*

The use of game theory in the semantics of programming languages is based upon work done by P. Lorenzen in the 1950's on "dialogue games" (Lorenzen 1960 (for a survey, cf. Felscher 1986), however we will here consider a slight variation of the games considered there in order to make the connection to Game Semantics in the following subsection more explicit). There a sentence of the propositional calculus (these are the formulae inductively constructed from atomic propositions p using the connectives $\vee, \wedge, \Rightarrow, \neg$) is interpreted via a two-player game between the "Proponent" trying to prove an assertion and the "Opponent" trying to disprove it. This is done recursively on the structure of the given formula. It can be done both for intuitionistic (in this logic essentially the law of the excluded third is not required to hold) and classical logic and we will start by giving the rules for the treatment of the former.

To formulate this more formally, we first need to be a little more precise: The players are called "**1**" and "**2**", and at each point in the game, each of them can either attack or defend the (sub-)formula under consideration at that point.

Thus the possible moves are: A player (**1** or **2**) can

- assert a formula (e.g., $A \vee B$) or
- attack a (previously asserted) formula (in the notation employed below this will be denoted by a "?" under the attacked formula).

A play of a game then is a sequence of moves made in turns by the two players **1** and **2** according to the following rules.

- **1** starts by asserting a formula and then it is **2**'s turn to move as the "attacker".
- If the player whose turn it is to move is currently in the attacker role he can attack the formula ϕ asserted by the other player in the preceding move in the following way:
  - If the currently attacked formula is of the form $A \wedge B$ he can attack one of the subformulas A or B (and moreover, he can later attack the other not yet attacked subformula).
  - If the currently attacked formula is of the form $A \vee B$, $A \Rightarrow B$ or $\neg A$ then he can simply attack the whole formula.
- If in the preceding move one of the players attacked the formula ϕ, the other one can now make the following moves (in "defense" of ϕ) depending on the structure of ϕ:

atomic If ϕ is an atomic formula, this depends on which player currently is to move: **2** can assert ϕ, but **1** can only assert ϕ if **2** has previously asserted it.

$A \wedge B$ He can simply assert the whole formula $A \wedge B$.

$\phi = A \vee B$ He can either assert A or B (under the proviso of the previous case if the chosen formula is atomic). (Note that however in this intuitionistic case he does not later have the option to also assert the other disjunct !)

$A \Rightarrow B$ He can attack A. Instead (or also additionally, at a later point of the game) he can assert B (under the above proviso).

$\neg A$ He can attack A.

- If in each of the subplays the player in turn cannot move then the play stops. If there is an attack that could not be answered, this can only be an attack by **2**, since **2** can assert atomic formulae ad libitum, and then **2** is the winner, and otherwise **1** is.

Note that the asymmetry of the rules wrt. atomic propositions comes from the fact that in order to show that a formula is valid (semantically), one has to show that it evaluates to true for any possible valuation of the atomic propositions involved.

These rules are pictured schematically in the following table:

1	$A \vee B$	**1**	$A \wedge B$
2	?	**2**	attack A or B
1	choose A or B	**1**	defend chosen formula
2	attack chosen formula		
1	$A \Rightarrow B$	**1**	$\neg A$
2	?	**2**	?
1	attack A or defend B	**1**	attack A

A player has won a single play of the game corresponding to a formula if he made the last move that is allowed according to the above rules (i.e., which is "legal"). A strategy for a player p (say **1**) is a function that assigns to every sequence of legal moves ending with a move of the other player (**2**) a move of his own (i.e., of **1**). Thus one can obtain a play of a game by playing a strategy for **1** off against a strategy for **2**. A strategy is said to be a winning strategy if this is done in such a way that p wins every possible play of the game. It then follows that winning strategies for **1**

asserting a formula ϕ correspond exactly to proofs of ϕ in intuitionistic logic (in short we have the slogans "propositions-as-assertion-moves" and "proofs-as-winning-strategies").

For illustration we present proofs via games for two formulae (here and in the following we present games by showing a typical run instead of all possible runs to increase readability):

Modus Ponens:

1	$((A$	$\Rightarrow$	$B)$	$\wedge$	$A)$	$\Rightarrow$	B							
2												?		
1			?											
2	?													
1									?					
2									A					
1	A													
2				B										
1												B		

Identity:

1	A	$\Rightarrow$	A
2		?	
1	?		
2	A		
1			A

Let us go through the game for the first formula in detail: Here **1** starts by asserting the formula $\phi = ((A \Rightarrow B) \wedge A) \Rightarrow B$. By the rules given above and the structure of ϕ the only move **2** can then do is to attack the whole formula. **1** can then defend ϕ by either attacking the premiss $(A \Rightarrow B) \wedge A$ (and possibly asserting the conclusion B later), or by asserting B immediately. According to the strategy represented by the above diagram she chooses the first option, which means that she must actually attack one of the conjuncts $A \Rightarrow B$ and A, and so she attacks the first (only to attack the second later – in fact she would also succeed by doing it the other way around). Now **2** can either attack the premiss A, or simply assert B (note that the latter option would not exist for **1** by the above rules). In the latter case **1** would immediately have the winning move to assert the conclusion B of ϕ (which is then possible because **2** has asserted it first). Thus in this play **2** chooses instead to attack A. Now **1** makes use of her still existing option to attack the other conjunct A (that she has not attacked before) of the conjunction $(A \Rightarrow B) \wedge A$ that has previously been under her attack. **2** can only defend A by simply asserting it. But then also **1** is allowed to defend A as the premiss of $A \Rightarrow B$ by asserting it. The only option left to **2** is to finish off her defense of $A \Rightarrow B$ by asserting B. But then again **1** may assert B as the conclusion of ϕ. Since there is no move left for **2**, **1** wins this play, and in fact from the explanations given above it is clear that **1** has a winning strategy for this game. Thus we have given a proof of ϕ in intuitionistic logic.

We obtain a representation of proofs of classical logic if the above rules are weakened so that not only both conjuncts of a conjunction can subsequently be attacked, but also both disjuncts of a disjunction subsequently be defended:

1	A	$\vee$	$\neg$	A
2		?		
1			$\neg$	A
2			?	
1				?
2				A
1	A			

2.2. *Game Semantics*

To lead over from Lorenzen Games to games in the semantics of programming languages we can make use of the Curry-Howard-Isomorphism. The idea is to view "propositions as types" and proofs for a proposition A as terms of type A in the λ-calculus (thus a proposition is interpreted as valid iff the corresponding type is inhabited). Recalling the above slogans this gives us "assertions-as-types" and "winning-strategies-as-terms" (note that for P's strategies to be winning means that the corresponding term denotes a total function). So for example the natural number $3 \in \mathbf{N}$ is represented by the following strategy:

1	**N**
2	q
1	3

Here we change our notation slightly to indicate the change of perspective: Firstly, instead of ? we write q. This is now interpreted as a request by the environment for an element of **N**. The difference to the preceding situation is that here our types are usually inhabited (while there the propositions were not always valid) and so the attention is turned from provability (the existence of *some* proof) to a specific proof. This we indicate by naming the proof (i.e., the element representing it – 3 in the above example) instead of simply asserting the proposition. Because of the fundamental difference in the interpretation in this setting of the first move (which asserts the type of the game) from the others we will not consider it as a move here, such that the game starts with a question by **2** (this player is here renamed to E for "environment", while **1** becomes S for "system").

The interpretation of a question and its corresponding answer is then the delivery of the requested data by the system to the environment. This has the consequence that a player can repeatedly "attack" the same $\rightarrow$-connective (which was not possible in Lorenzen Games) in order to get different inputs (see the examples below).

Here "system" and "environment" can take several interpretations: for instance, a computer system and its user, a computer and the other computers of its network, or in the program text a term and its context. The explicit distinction between system and environment from the beginning is an important difference to most other process models. (In CSP, for example, one does have two different operators for internal and external choice. On the other hand, Hoare takes the view that: "In choosing an alphabet, there is no need to make a distinction between events which are initiated by the object (perhaps *choc*) and those which are initiated by some agent outside the object (for example, *coin*). The avoidance of the concept of causality leads to considerable simplification in the theory and its application" (Hoare 1985, 24).)

Note also that a value (that in standard denotational semantics is atomic) is here represented by an interactive process ("splitting the atom of computation").

Under the Curry-Howard-Isomorphism, $\Rightarrow, \wedge, \vee$, **true**, **false** correspond to the function space $\rightarrow$, the cartesian product $\times$, the disjoint sum $+$, the singleton 1 and the empty set $\emptyset$ respectively (and so we will adopt the latter notation).

The above example Identity here instantiates to the identity function of type $A \rightarrow A$ (represented by the "copycat-strategy"):

	A	$\rightarrow$	A
E			q
S	q		
E	a		
S			a

(Note that for clarity we write the request under the corresponding type, and not under the connective $\rightarrow$ as in Lorenzen Games.) Similary, Modus Ponens corresponds to function application.

To give a few more examples:

Addition:

	N	→	(**N**	→	**N**)
E					q
S	q				
E	n				
S			q		
E			m		
S					$n+m$

$\lambda f : \mathbf{N} \to \mathbf{N}.\text{if } f(0) = 0 \text{ then 1 else 0}$

	(**N**	→	**N**)	→	**N**
E					q
S			q		
E	q				
S	0				
E			$\begin{smallmatrix} n=0 \\ n>0 \end{smallmatrix}$ or		
S					$\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}$ or

The first example is a particular run of the strategy that, after being requested an output by the environment, itself requests the two input arguments, and then returns their sum. Note that the same function could be modelled by a different strategy, namely by the one that takes the arguments in the reverse order. This illustrates the intensionality of this model, and it in fact models the situation correctly also if one takes into account features of actual computation like store and control (see below).

The second one is an example for a higher-order function: it pictures a strategy for the function

$$g := (\lambda f : \mathbf{N} \to \mathbf{N}.\text{if } \mathbf{f(0)} = \mathbf{0} \text{ then 1 else 0}) : (\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$$

that takes a function $f : \mathbf{N} \to \mathbf{N}$ and returns 1 if the function has value 0 at input 0, otherwise 0. Note that g receives its input not "all at once" (as a first-order function would receive its input, e.g., a natural number), but in a "demand-driven" fashion. This is in accordance with the general way of representation by (finitary) interaction and is necessary to satisfactorily model programming languages, since on a computer one cannot deal directly with infinite objects (like functions with infinite domain), but only indirectly through a finite representation by a term or (as here) finite (but arbitrary) portions of it. Thus after being requested an output from the environment, g requests a value from its input f. In this run of the strategy, f in turn demands from g a value (its argument) and receives 0, whereupon it delivers its value at 0. If this value equals 0, g returns 1, otherwise 0 to the initial request (for simplification these two cases are depicted in the same diagram, so in fact the diagram represents to possible runs: the first one is obtained by substituting the first instances $n = 0$ and 1 in the last two lines, and the second one by using the second cases $n > 0$ and 0.

Note that one can also model non-strict functions (a function is non-strict if it delivers a defined output even for an undefined input, in

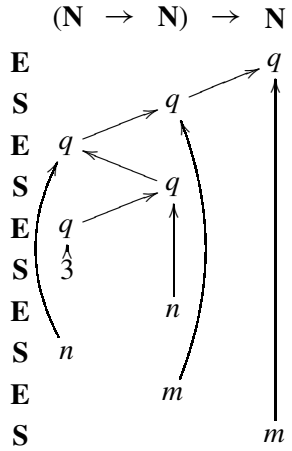
the domain-theoretic sense): The following function delivers 3 without looking at its input:

	N	$\rightarrow$	**N**
E			q
S			3

By the above remarks about inputs to higher-order functions, one often requires several interactions between the function and its argument, as in the following example of the function $\lambda f.f(0) + f(1) : (\mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{N}$ that takes a function $f : \mathbf{N} \rightarrow \mathbf{N}$ as input and outputs the value $f(0) + f(1) \in \mathbf{N}$ (note that here we deviate from Lorenzen Games by allowing the same connective to be "attacked" twice, as indicated above):

	(N	$\rightarrow$	**N)**	$\rightarrow$	**N**
E					q
S			q		
E	q				
S	0				
E			n		
S			q		
E	q				
S	1				
E			m		
S				$n+m$	

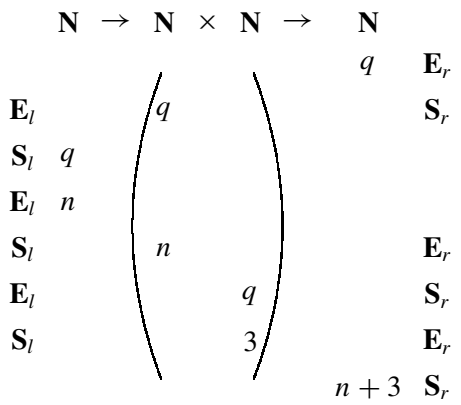
These interactions can also be nested in each other, as in the following example of the function $\lambda f.f(f(3)) : (\mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{N}$ that takes a function $f : \mathbf{N} \rightarrow \mathbf{N}$ as input and produces the value $f(f(3)) \in \mathbf{N}$ (where we introduce pointers to indicate which question provided data refer to; this concept is defined more precisely in the next section):



Whereas in the example above one would not really need the pointers by instead making the convention that each delivered data refers to the "pending" question (this is the "well-bracketing" condition defined below), there are more complicated examples where this is not possible: For example, $\lambda f.f(\lambda x.f(\lambda y.y))$ and $\lambda f.f(\lambda x.f(\lambda y.x))$ would be identified without the use of pointers.

In modelling systems it is conceptually nice (and for more complicated systems even required in order to make modelling feasible) to model the different components and then obtain a model of the whole by putting together the models of the parts. In order to do this one needs to be able to compose the strategy representing one component (which in the above system/environment-distinction takes on the role of the "system") with the strategy representing the joint behaviour of the other components (the "environment" of the former component).

To visualize composition of (i.e., interaction between) strategies, consider the following example of the composition of $\lambda n.(n, 3) : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ (that maps a value n to $(n, 3)$) followed by the (uncurryed) addition $+ : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$:



The interaction takes place in the following way: The play starts in the right game with the question from $\mathbf{E}_r$ in $\mathbf{N}$. According to the strategy for $+$ this prompts a question from $\mathbf{S}_r$ in the left factor of $\mathbf{N} \times \mathbf{N}$. Now any question by $\mathbf{S}_r$ in the domain of the strategy on the right (and thus in the codomain of the strategy on the left) is in the game on the left interpreted as a move by $\mathbf{E}_l$. This corresponds very nicely to the intuitive fact that every component of a whole is part of the environment for any other component. Now according to the strategy for $\lambda n.(n, 3)$, since $\mathbf{E}_l$ asks for the left factor of $\mathbf{N} \times \mathbf{N}$, this results in a demand of input in $\mathbf{N}$ by $\mathbf{S}_l$, and the response n by $\mathbf{E}_l$ is copied to the left factor of $\mathbf{N} \times \mathbf{N}$. There it is interpreted as a move by $\mathbf{E}_r$ and the game continues similarly. In the end, the strategy resulting from the composition is obtained by "hiding" the moves that are not any longer in interaction with the overall environment (the ones in brackets in the middle). As expected, the corresponding function is $\lambda n.n + 3$.

NOTE 1.

- Note that as with composition of functions one can only compose strategies with matching types: to form $\sigma; \tau$, we must have σ of the type $A \to B$ and τ of the type $B \to C$ for suitable A, B, C.
- In the setting of Lorenzen Games, composition of strategies gives us a natural proof for the transitivity of $\Rightarrow$.
- Obviously there is a close relationship of the composition of strategies to the "parallel composition + hiding" in the process algebra CSP (because of the way our strategies are typed it is here possible to put these two constructions together without losing associativity; with the same idea one can also construct typed processes (cf., e.g., Abramsky (1996); Jürjens (1999b)).

- In addition to associativity we also have (partial) neutral elements wrt. composition (given by the identity strategies as presented above), so in fact we can form a category (see below).

As a special case of composition we get the application of strategies to their input: Define I to be the empty game with no moves (and one strategy, namely the one that does nothing). Then we can represent, e.g., the element $(3, 5) \in \mathbf{N} \times \mathbf{N}$ by the strategy

$$\begin{array}{ccccc} I & \to & \mathbf{N} & \times & \mathbf{N} \\ & & q & & \\ & & 3 & & \\ & & & & q \\ & & & & 5 \end{array}$$

and so $3 + 5$ becomes in fact 8:

$$\begin{array}{ccccccccc} & I & \to & \mathbf{N} & \times & \mathbf{N} & \to & \mathbf{N} & \\ & & & & & & & q & \mathbf{E}_r \\ \mathbf{E}_l & & & q & & & & & \mathbf{S}_r \\ \mathbf{S}_l & & & 3 & & & & & \mathbf{E}_r \\ \mathbf{E}_l & & & & & q & & & \mathbf{S}_r \\ \mathbf{S}_l & & & & & 5 & & & \mathbf{E}_r \\ & & & & & & & 8 & \mathbf{S}_r \end{array}$$

With game semantics one can also model the key ingredients of imperative languages, namely commands and store, and furthermore one can define control operators that allow early escape from function evaluation. One of the nicest features of game semantics is that the abilities to use store resp. control correspond exactly to different kinds of internal properties of the strategies involved (this will be made more precise below).

3. DEFINITIONS

To put the above intuitive examples on a more solid foundation, we will now provide the underlying definitions. They appeared in McCusker (1998) and are essentially an adaption of Hyland and Ong (2000), taking account of ideas in Abramsky et al. (2000).

3.1. *Games and Strategies*

DEFINITION 1. An *arena* is a structure $A = (M_A, \lambda_A, \vdash_A)$ consisting of

- a set of *moves* M_A,
- the labelling function $\lambda_A : M_A \to \{\mathbf{S}, \mathbf{E}\} \times \{Q, A\}$ (call moves labelled $(\mathbf{S}, l)$ resp. $(\mathbf{E}, l)$ (for $l \in \{Q, A\}$) "**S**-moves" resp. "**E**-moves" and moves labelled (l, Q) resp. (l, A) (for $l \in \{E, S\}$) "questions" resp. "answers") and
- the enabling relation $\vdash_A \subseteq (M_A + \{\iota\}) \times M_A$ (with $\iota \notin M_A$; say m enables n if $m \vdash n$ — the idea is that during a play moves can be made only when they are enabled by earlier moves.). Call a move that is enabled by ι "initial".

under the following conditions:

- *Initial* moves are **E**-questions, and they are not enabled by any other moves besides ι.
- Answers can only be enabled by questions.
- Enabling alternates between **E**-moves and **S**-moves (i.e., an **E**-move can only enable a **S**-move and vc. vs.).

DEFINITION 2.

- A *justified sequence* is a sequence s of moves together with each a justification pointer from every non-initial move m to a move n earlier in s such that $n \vdash m$. We say that (this occurrence of) the move n justifies m and write this as $n \overleftarrow{\cdot t \cdot} m$
  (where $\cdot$ denotes concatenation, and supposing that t is the subsequence of moves between n and m).Note that justified sequences always start with **E**-questions.
- For a justified sequence s, we define the *system view* $\overline{s}$ and the *environment view* $\underline{s}$ of s by induction on the length of s:

$$
\begin{aligned}
\overline{\varepsilon} &= \varepsilon. \\
\overline{s \cdot m} &= \overline{s} \cdot m, \text{ if } m \text{ is a } \mathbf{S}\text{-move.} \\
\overline{s \cdot m} &= m, \text{ if m is initial.} \\
\overline{s \cdot m \overleftarrow{\cdot t \cdot} n} &= \overline{s} \cdot m \overleftarrow{\cdot} n, \text{ if } n \text{ is an } \mathbf{E}\text{-move.} \\
\underline{\varepsilon} &= \varepsilon. \\
\underline{s \cdot m} &= \underline{s} \cdot m, \text{ if } m \text{ is an } \mathbf{E}\text{-move.} \\
\underline{s \cdot m \overleftarrow{\cdot t \cdot} n} &= \underline{s} \cdot m \overleftarrow{\cdot} n, \text{ if } n \text{ is an } \mathbf{E}\text{-move.}
\end{aligned}
$$

- A justified sequence s is a *legal position* if
  - players alternate (if $s = s_1 \cdot m \cdot n \cdot s_2$ and m is an **E**-move, then n is a **S**-move and vc. vs.) and
  - for any prefix $t \cdot m$ of s: if m is a **S**-move, then its justifier is in $\overline{t}$ and if m is a non-initial **E**-move then its justifier is in $\underline{t}$.

  Write L_A for the set of legal positions of A.

DEFINITION 3.

- Let m be a move in a legal position s. We say that m is *hereditarily justified* by an occurrence of a move n in s if there is a subsequence of s starting with n and ending in m such that every move is justified by the preceding move in it. For a set of (occurrences of) initial moves we write $s\lceil I$ for the subsequence of s consisting of the moves hereditarily justified by a move of I.
- A *game* is a structure $A = (M_A, \lambda_A, \vdash_A, P_A)$ where
  - $(M_A, \lambda_A, \vdash_A)$ is an arena and
  - P_A is a non-empty, prefix-closed subset of L_A called the *valid positions* such that for $s \in P_A$ and I a set of initial moves of s we have $s\lceil I \in P_A$.
- A (deterministic) *strategy* σ for a game A is a non-empty set of even-length positions from P_A satisfying
  - $s \cdot a \cdot b \in \sigma \Rightarrow s \in \sigma$ and
  - $s \cdot a \cdot b, s \cdot a \cdot c \in \sigma \Rightarrow b = c$ and b and c have the same justifier (determinacy condition).

3.2. *Composition of Strategies*

Now we would like to model compositionality. It is convenient to do this in the framework of category theory, because that way we can make use of already existing results on models of PCF (or linear logic).

A category consists of objects and morphisms. The objects of our category will be the games. To define the notion of morphism we first need to consider a construction on games:

DEFINITION 4. Given games A and B, the game $A \multimap B$ is defined as follows ($A \multimap B$, as opposed to $A \to B$, is the usual notation for the morphisms sets in models of linear logic):

$$M_{A \multimap B} = M_A + M_B$$

$$\begin{aligned}
\lambda_{A \multimap B} &= [\bar{\lambda}_A, \lambda_B] \\
\iota \vdash_{A \multimap B} m &\Leftrightarrow \iota \vdash_B m \\
m \vdash_{A \multimap B} n &\Leftrightarrow m \vdash_A n \vee m \vdash_B n \ \vee [\iota \vdash_B m \wedge \iota \vdash_A n] \text{ for m} \neq \iota \\
P_{A \multimap B} &= \{s \in L_{A \multimap B} : s \mid_A \in P_A \wedge s \mid_B \in P_B\}.
\end{aligned}$$

(where $\bar{\lambda}_A$ means λ_A with the **S/E**-labels inverted and $s|_A$ is the subsequence of s consisting of moves from M_A).

Now a morphism from a game A to a game B is a strategy on $A \multimap B$. After some auxiliary definitions we will give the definition of composition of strategies:

- For a sequence u of moves from games A, B, C with justification pointers define $u \mid_{B,C}$ to be the subsequence of u consisting of moves from B and C (removing pointers that point to moves from A). Similarly define $u|_{A,B}$. u is an *interaction sequence* of A, B, C if $u \mid_{A,B} \in P_{A \multimap B}$ and $u \mid_{B,C} \in P_{B \multimap C}$. Write the set of all such sequences as $\text{int}(A, B, C)$.
- Suppose $u \in \text{int}(A, B, C)$. By definition of $\multimap$, a pointer from an A-move a can only point to a B-move b if b is initial and its pointer points to an initial C-move c. Define $u \mid_{A,C}$ to be the subsequence of u consisting of the moves of A and C where in the mentioned case the pointer from a is changed to point to c.
- Given strategies $\sigma : A \multimap B$, $\tau : B \multimap C$, define

  $$\sigma \| \tau := \{u \in \text{int}(A, B, C) : u \mid_{A,B} \in \sigma \wedge u \mid_{B,C} \in \tau\}$$

  and finally the composition of σ followed by τ to be

  $$\sigma; \tau := \{u \mid_{A,C} : u \in \sigma \| \tau\}.$$

PROPOSITION 1. We obtain a category $\mathcal{G}$ whose objects are games and where the morphisms from A to B are strategies $\sigma : A \multimap B$ with composition as defined above and identities the copycat-strategies.

3.3. *Restrictions on Strategies*

In this section we will define certain restrictions on the sets of strategies that are needed for the game-semantical characterization of programming disciplines mentioned earlier.

DEFINITION 6. By determinacy of strategies we know that for $s \cdot a \cdot b, t \cdot a \in L_A$ (where $s \cdot a \cdot b$ has even length) with $\overline{s \cdot a} = \overline{t \cdot a}$, there is a

unique (by determinacy) extension $\text{match}(s \cdot a \cdot b, t \cdot a)$ of $t \cdot a$ by b (with a justification pointer for b) such that $\overline{s \cdot a \cdot b} = \overline{\text{match}(s \cdot a \cdot b, t \cdot a)}$. A strategy σ on A is called *innocent* iff in each such situation it satisfies

$$\begin{aligned} s \cdot a \cdot b \in \quad & \sigma \wedge t \in \sigma \wedge t \cdot a \in P_A \wedge \overline{t \cdot a} = \overline{s \cdot a} \\ \Rightarrow \ & \text{match}(s \cdot a \cdot b, t \cdot a) \in \sigma, \end{aligned}$$

i.e., a move by **S** depends only on the **S**-view.

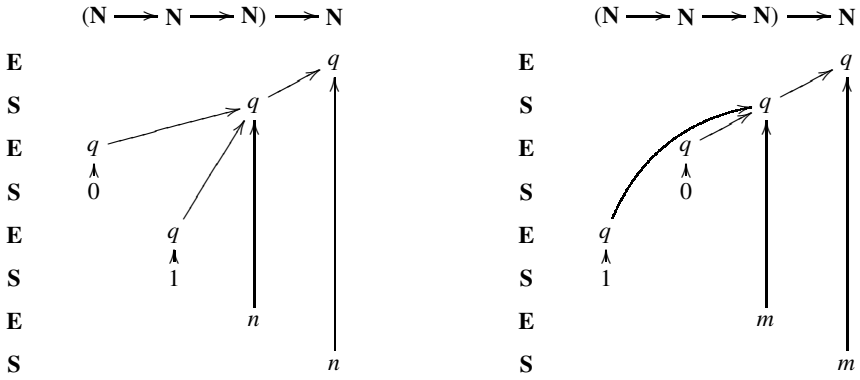
For an example for a non-innocent strategy consider the following function (strictly speaking, the following examples are strategies in the category $\mathcal{C}$ to be derived from $\mathcal{G}$ in the next subsection):

$$\begin{aligned} F \ := \ & \lambda f : \mathbf{N} \to (\mathbf{N} \to \mathbf{N}). \\ & \text{new } x := 0 \text{ in } f(\text{if } x = 0 \text{ then } (x := 1; 0) \text{ else } 1) \\ & \qquad\qquad\qquad\qquad (\text{if } x = 0 \text{ then } (x := 1; 0) \text{ else } 1) \end{aligned}$$

Then we have

$$Ff = \begin{cases} f\ 0\ 1, & \text{if } f \text{ asks for its first argument first} \\ f\ 1\ 0, & \text{if } f \text{ asks for its second argument first} \end{cases}$$

The strategy for this function has the following two runs:

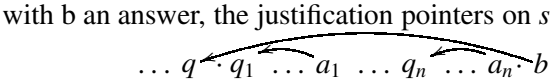


This violates innocence: Since

$$\overline{q_1 \overset{\frown}{\cdot} q_2 \overset{\frown}{\cdot} q_3} = q_1 \overset{\frown}{\cdot} q_2 \overset{\frown}{\cdot} q_3 = \overline{q_1 \overset{\frown}{\cdot} q_2 \cdot q_4 \overset{\frown}{\cdot} 0 \cdot q_3},$$

P must do the same in both runs.

DEFINITION 7. A strategy σ is *well-bracketed* iff for each $s \cdot a \cdot b \in \sigma$ with b an answer, the justification pointers on $s \cdot a \cdot b$ have the form

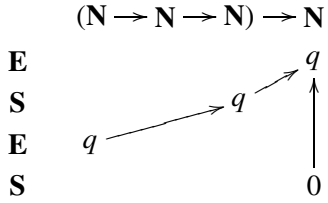


(with $a_n = a$ and where a_i are answers), i.e., **S** can answer only the most recent unanswered question in **S**'s view.

A counter-example for the well-bracketing condition is provided by the control operator catch: $(\mathbf{N} \to (\mathbf{N} \to \mathbf{N})) \to \mathbf{N}$ which is defined by

$$\text{catch}(f) = \begin{cases} 0, \text{ if } f \text{ calls its first argument first,} \\ 1, \text{ if } f \text{ calls its second argument first,} \\ n+2, \text{ if } f \text{ returns } n \text{ immediately.} \end{cases}$$

The following is a possible run for the corresponding strategy, where clearly the bracketing condition is violated:



PROPOSITION 2. We obtain categories $\mathcal{G}_i$, $\mathcal{G}_b$ resp. $\mathcal{G}_{ib}$ that are subcategories of $\mathcal{G}$ with the same objects and morphisms the innocent, well-bracketed resp. innocent and well-bracketed strategies.

3.4. *Cartesian Closedness*

Models of the lambda-calculus (and so in particular of PCF) are often given in the framework of Cartesian closed categories (ccc's).

Note that all four defined categories are autonomous, i.e., symmetric monoidal closed (and this structure is respected by the subcategory inclusions), via the following tensor product (and the unit $I = (\emptyset, \emptyset, \emptyset, \{\varepsilon\})$):

DEFINITION 8. Given games A and B, the game $A \otimes B$ is defined as follows:

$$\begin{aligned} M_{A\otimes B} &= M_A + M_B \\ \lambda_{A\otimes B} &= [\lambda_A, \lambda_B] \\ \iota \vdash_{A\otimes B} m &\Leftrightarrow \iota \vdash_A m \vee \iota \vdash_B m \\ m \vdash_{A\otimes B} n &\Leftrightarrow m \vdash_A n \vee m \vdash_B n \\ P_{A\otimes B} &= \{s \in L_{A\otimes B} : s \mid_A \in P_A \wedge s \mid_B \in P_B\}. \end{aligned}$$

(where $\bar{\lambda}_A$ means λ_A with the **S/E**-labels inverted and $s|_A$ is the subsequence of s consisting of moves from M_A).

We will make use of the autonomous structure in order to obtain cartesian closed categories out of the categories defined above using the Girard translation of intuitionistic logic into linear logic. First we will define the categorical product.

DEFINITION 9. Given games A and B, the game $A \times B$ is defined as follows:

$$\begin{aligned}
M_{A\times B} &= M_A + M_B \\
\lambda_{A\times B} &= [\lambda_A, \lambda_B] \\
\iota \vdash_{A\times B} m &\Leftrightarrow \iota \vdash_A m \vee \iota \vdash_B m \\
m \vdash_{A\times B} n &\Leftrightarrow m \vdash_A n \vee m \vdash_B n \\
P_{A\times B} &= \{s \in L_{A\times B} : s\,|_A \in P_A \wedge s\,|_B = \varepsilon\} \\
&\quad \cup \{s \in L_{A\times B} : s\,|_B \in P_B \wedge s\,|_A = \varepsilon\}.
\end{aligned}$$

The projections are the obvious copycat strategies.

It is straightforward to generalize the definition from the binary to the set-indexed case and to show that this actually gives a categorical product.

To define the morphisms in the ccc's to be constructed we need the exponential of a game:

DEFINITION 10. Given a game A, the game $!A$ is defined as follows:

$$\begin{aligned}
M_{!A} &= M_A \\
\lambda_{!A} &= \lambda_A \\
\vdash_{!A} &= \vdash_A \\
P_{!A} &= \{s \in L_{!A} \mid \text{for each initial move } m, s\lceil m \in P_A\}.
\end{aligned}$$

Intuitively, $!A$ stands for arbitrarily many copies of A. The use of this operator is necessitated by the fact that the λ-calculus, as opposed to linear logic, is not resource-sensitive.

To define composition of morphisms $\sigma : A \to B$ in $\mathcal{C}$ (which will be strategies $!A \multimap B$ in $\mathcal{G}$) with $\tau : B \to C$ we will then need for each strategy $\sigma : !A \multimap B$ a "lifting" $\sigma^\dagger : !A \multimap !B$. This, however, can only be defined for a restricted class of games:

DEFINITION 11. A game A is *well-opened* iff for all $sm \in P_A$ with m initial, $s = \varepsilon$.

For $\sigma : !A \multimap B$ with well-opened games A, B define $\sigma^\dagger : !A \multimap !B$ by

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial } m, s\lceil m \in \sigma\}.$$

One can show that for well-opened games this construction does not only preserve the property of being a strategy, but also that of being innocent and well-bracketed.

Now we can construct a ccc from each of the categories defined above using the Girard translation:

DEFINITION 12. The category $\mathcal{C}$ has as objects well-opened games and as morphisms $\sigma : A \to B$ strategies for $!A \multimap B$. The composition $\sigma; \tau : A \to C$ of morphisms $\sigma : A \to B$ and $\tau : B \to C$ is defined to be $\sigma^\dagger; \tau$. The subcategories $\mathcal{C}_i$, $\mathcal{C}_b$ and $\mathcal{C}_{ib}$ are defined by imposing restrictions analogously to the definitions above.

One can show that each of these four categories is cartesian closed and that this additional structure is respected by the inclusions. As usual in ccc's let us write $(A \Rightarrow B) := (!A \multimap B)$, $\Lambda(f) : A \to (B \Rightarrow C)$ for the morphism obtained by currying $f : A \times B \to C$, and $\mathrm{ev} : (A \Rightarrow B) \times A \to B$ for the morphism obtained by uncurrying the identity on $A \Rightarrow B$.

In fact there are conceptually very appealing factorization theorems that show that each strategy can be factored into an innocent (resp. well-bracketed) and a non-innocent (resp. non-well-bracketed) part.

4. FULLY ABSTRACT MODELS FOR PROGRAMMING LANGUAGES

In the following we will present the basic results about game-semantical models for programming languages. We start by defining the language in question.

4.1. *The Language PCF*

The programming language PCF is a call-by-name functional language with a base type of expressions denoting natural numbers and constants for arithmetic and recursion. Its syntax is that of an applied simply-typed λ-calculus (for a definition of λ-calculus cf. Matthes (2002)) with types given by the following grammar:

$$A ::= \mathsf{exp} \mid A_1 \to A_2.$$

Terms are defined as follows:

$$\begin{aligned} M \ ::= \ & x \mid \lambda x : A.M \mid M_1 M_2 \\ & \mid n \mid \mathsf{succ} M \mid \mathsf{pred} M \\ & \mid \mathsf{cond} M_1 M_2 M_3 \mid \mathsf{Y}_A M \end{aligned}$$

(where x is a variable and n a natural number).

Typing judgements are made using the following rules:

Variables: $\dfrac{}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i} i \in \{1, \ldots, n\}$

Functions: $\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}, \dfrac{\Gamma \vdash M : A \to B, \Gamma \vdash N : A}{\Gamma \vdash MN : B}$

Arithmetic: $\dfrac{}{\Gamma \vdash n : \mathsf{exp}}, \dfrac{\Gamma \vdash M : \mathsf{exp}}{\Gamma \vdash \mathsf{succ} M : \mathsf{exp}}, \dfrac{\Gamma \vdash M : \mathsf{exp}}{\Gamma \vdash \mathsf{pred} M : \mathsf{exp}}$

Conditional and recursion: $\dfrac{\Gamma \vdash M \mathsf{exp}, \Gamma \vdash N_1 : \mathsf{exp}, \Gamma \vdash N_2 \mathsf{exp}}{\Gamma \vdash \mathsf{cond} M N_1 M_2 : \mathsf{exp}}$,

$$\frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathsf{Y}_A M : A}$$

The "big-step" operational semantics of PCF is given by a relation $M \Downarrow V$ ("M evaluates to V") where M is a closed term (a term with no variables, i.e., so that $\vdash M : A$ can be derived) and V is a *canonical form* defined by the following grammar:

$$V ::= n \mid \lambda x.M$$

This determines a partial function from closed term of type exp to natural numbers in the following way (where $M[N/x]$ is the capture-free substitution of the term N for the variable x in the term M):

Canonical forms: $\dfrac{}{V \Downarrow V}$

Functions: $\dfrac{M \Downarrow \lambda x.M',\ M'[N/x] \Downarrow V}{MN \Downarrow V}$

Arithmetic: $\dfrac{M \Downarrow n}{\mathsf{succ}M \Downarrow n+1}, \dfrac{M \Downarrow n+1}{\mathsf{pred}M \Downarrow n}, \dfrac{M \Downarrow 0}{\mathsf{pred}M \Downarrow 0}$

Conditional: $\dfrac{M \Downarrow 0,\ N_1 \Downarrow V}{\mathsf{cond}MN_1N_2 \Downarrow V}, \dfrac{M \Downarrow n+1,\ N_2 \Downarrow V}{\mathsf{cond}MN_1N_2 \Downarrow V}$

Recursion: $\dfrac{M(\mathsf{Y}M) \Downarrow V}{\mathsf{Y}M \Downarrow V}$

.

4.2. *Game-semantical Characterization of Programming Disciplines*

We will first give the usual interpretation of the simply-typed λ-calculus in a Cartesian closed category. For $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ let us write $[\![\Gamma]\!] := [\![A_1]\!] \times \ldots \times [\![A_n]\!]$.

Each type A is modelled by an object $[\![A]\!]$: Starting with the definition of $[\![\exp]\!]$ (see below), higher types are defined by $[\![A \to B]\!] = [\![A]\!] \Rightarrow [\![B]\!]$.

A term $\Gamma \vdash M : A$ is modelled as a morphism $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$:

Variables are interpreted by projections:

$$[\![\Gamma \vdash x_i : A_i]\!] = \pi_i : [\![\Gamma]\!] \to [\![A_i]\!].$$

Abstraction is modelled by currying:

$$[\![\Gamma \vdash \lambda x : A.M : A \to B]\!]$$

$$= \Lambda([\![\Gamma, x : A \vdash M : B]\!]) : [\![\Gamma]\!] \to [\![A]\!] \Rightarrow [\![B]\!].$$

Application is interpreted via the evaluation map $\mathrm{ev} : (A \Rightarrow B) \times A \to B$:

$$[\![\Gamma \vdash MN : B]\!] = ([\![\Gamma \vdash M : A \to B]\!], [\![\Gamma \vdash N : A]\!]); \mathrm{ev}.$$

Thus to obtain a model for PCF in any of the four ccc's defined above we are left to interpret the type exp and the term constants n, succ$M$, pred$M$, condMN_1N_2 and $\mathrm{Y}_A M$:

$[\![\mathrm{exp}]\!]$ is the flat game $\mathbf{N}$ of natural numbers:

$$\begin{aligned} M_{\mathbf{N}} &= \{q\} \cup \{n \mid n \in \omega\} \\ \lambda_{\mathbf{N}}(q) &= \mathrm{OQ} \\ \lambda_{\mathbf{N}}(n) &= \mathrm{PA\ (for\ each\ } n) \\ \iota &\vdash_{\mathbf{N}} q \\ q &\vdash_{\mathbf{N}} n\ \mathrm{(for\ each\ } n) \\ P_{\mathbf{N}} &= \{\varepsilon, q\} \cup \{qn \mid n \in \omega\} \end{aligned}$$

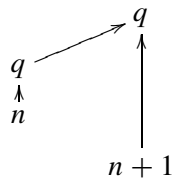
The strategies for $\mathbf{N}$ are $\perp = \{\varepsilon\}$ and $[\![n]\!] = \{\varepsilon, qn\}$ for each n.

The constant succ is interpreted as

$$[\![\Gamma \vdash \mathrm{succ}M : \mathrm{exp}]\!] = ([\![\Gamma \vdash M]\!]; s) : [\![\Gamma]\!] \to [\![\mathrm{exp}]\!].$$

using the morphism $s : [\![\mathrm{exp}]\!]$ represented by the following strategy:

$!\mathbf{N} \multimap \mathbf{N}$



The operation pred is defined similarly. The conditional is then defined as

$$[\![\Gamma \vdash \mathrm{cond}MN_1N_2]\!] = ([\![\Gamma \vdash M]\!], [\![\Gamma \vdash N_1]\!], [\![\Gamma \vdash N_2]\!]); c$$

using the morphism $c : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ represented (via the canonical isomorphism $!(\mathbf{N}\times\mathbf{N}\times\mathbf{N}) = !\mathbf{N}\otimes!\mathbf{N}\otimes!\mathbf{N}$) by the strategy whose two typical plays are depicted below:

$!\mathbf{N}$	$\otimes$	$!\mathbf{N}$	$\otimes$	$!\mathbf{N}$	$\multimap$	$\mathbf{N}$
						q
q						
0						
		q				
		n				
						n

$!\mathbf{N}$	$\otimes$	$!\mathbf{N}$	$\otimes$	$!\mathbf{N}$	$\multimap$	$\mathbf{N}$
						q
q						
$m \neq 0$						
				q		
				n		
						n

Finally recursion is interpreted in the usual way making use of the fact that the ccc's defined above are cpo-enriched.

One can then show that the categories $\mathbf{C_{ib}}$, $\mathbf{C_b}$ and $\mathbf{C_i}$ (or more precisely, their quotient by an intrinsic preorder on the hom-sets) via the above interpretations give fully abstract models for the languages PCF, (a simplified version of) Idealized Algol and (a minor variant of) SPCF resp. . Here Idealized Algol is viewed as an extension of PCF with the constructs of a basic imperative language and block-allocated variables. More precisely, we add the two base types **com** (for commands which alter the state and which can be composed sequentially) and **var** (for variables which store natural numbers, that are allocated using an operator **new** x **in** M and that can be written to and read from). SPCF is an extension of PCF by control operators. More precisely, this variant of it is obtained by adding to PCF a family of control operators $\mathbf{catch}_k$. Intuitively, $\mathbf{catch}_k\ x_1, \ldots, x_k$ **in** M terminates immediately when the term M tries to evaluate the variable x_i and returns $i - 1$. If M delivers n without using any of the x_i, $\mathbf{catch}_k\ x_1, \ldots, x_k$ **in** M returns $n + k$.

Thus one obtains the following semantic characterization of programming disciplines (Abramsky and McCusker 1997, 1999b; Laird 1997; Abramsky et al. 1998) (the last case *functional + store + control* has not been published yet):

Constraints	Language
D+I+B	purely functional
D+I	functional + control
D+B	functional + store
D	functional + store + control

Here D stands for the subcategory of **C** with the same objects and the morphisms restrained by the determinacy condition (resp. I by the innocence and B the well-bracketing condition).

5. FURTHER WORK

By further work the above results have been considerably extended and include recursive types (McCusker 1998) and call-by-value (Abramsky and McCusker 1999b; Honda and Yoshida 1997). Thus at least in principle the main features of languages like Scheme or Core ML (except for the ability to test references for equality) have been taken care of. Also there has been research towards nondeterminism (Harmer and McCusker 1999).

Very recently there has been developed a new concurrent form of game semantics resolving problems posed by the sequentiality of the traditional ones and giving a full completeness result for multiplicative-additive linear logic (Abramsky and Melliès 1999; Abramsky 1999b). Also, game semantics has been employed to develop a notion of "Process Realizability" (Abramsky 1999b).

Applications of game semantics to reasoning about security issues can be found in Malacaria and Hankin (1999). Some of the work currently in progress addresses subtyping, and in another line of research, game-semantical ideas are being employed in specification and refinement in a way that takes account of program dynamics and the system/environment distinction (Abramsky 1999a, Jürjens 1999a).

Further work will address semantics for object-oriented languages (Java) and logical principles for structuring protocols.

6. CONCLUSION

Since this paper was intended to be an elementary introduction to game semantics and just to convey the basic intuitions, many details had to be left out. For these the reader is referred to Abramsky (1997a) and Abramsky and McCusker (1999a).

ACKNOWLEDGEMENT

This work was supported by the Studienstiftung des deutschen Volkes.

The author is very grateful to his supervisor, Prof. Samson Abramsky, for teaching him the subject of this introduction. Material from Abramsky and McCusker (1999a) and Abramsky (1997b; 1999c) was used extensively for this paper.

Furthermore the author would like to thank Benedikt Löwe and Florian Rudolph for organizing the Research Colloquium "Foundations of the Formal Sciences" (where the talk on which this paper is based was delivered), and R. Matthes, S. Merz and the other participants for interesting discussions. Further thanks go to Andreas Seidl for comments on the draft and the anonymous referee for insightful suggestions. This work is dedicated to the author's mother on the occasion of her birthday.

REFERENCES

Abramsky, S.: 1999, 'Retracing Some Paths in Process Algebra', in *CONCUR '96: Concurrency Theory (Pisa)*, Springer Verlag, Berlin, pp. 1–17.

Abramsky, S.: 1997a, 'Semantics of Interaction: An Introduction to Game Semantics', in A. Pitts and P. Dybjer (eds.), *Semantics and Logics of Computation*, Cambridge, 1995), Cambridge, pp.1–31.

Abramsky, S.: 1997b, 'Games in the Semantics of Programming Languages', in P. Dekker, M. Stokhof and Y. Venema (eds.), *Proceedings of the 11th Amsterdam Colloquium*, ILLC, Department of Philosophy, University of Amsterdam 1997, pp. 1–6.

Abramsky, S.: 1999a, *A note on Reactive Refinement*.

Abramsky, S.: 1999b, 'Process Realizability/Concurrent Games & Full Completeness of Linear Logic', Lecture Notes for the Lectures at the Marktoberdorf Summer School.

Abramsky, S.: 1999c, 'Game Semantics and Full Abstraction for Sequential Programming Languages', Course at LFCS, University of Edinburgh.

Abramsky, S., Honda, K., and McCusker, G.: 1998, 'A Fully Abstract Game Semantics for General References', in *Proceedings of the Thirteenth International Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, pp. 334–344.

Abramsky, S. and Jagadeesan, R.: 1992, 'Games and Full Completeness for Multiplicative Linear Logic (extended abstract)', in R. Shyamsunder (ed.), *Foundations of Software Technology and Theoretical Computer Science*, New Delhi, 1992, Berlin, pp. 291–301.

Abramsky, S. and Jagadeesan, R.: 1994, 'Games and Full Completeness for Multiplicative Linear Logic', *Journal of Symbolic Logic* **59**, 543–574.

Abramsky, S., Jagadeesan, R., and Malacaria, P.: 1994, 'Full Abstraction for PCF (extended abstract)', in M. Hagiya and J. C. Mitchell (eds.), *Theoretical Aspects of Computer Software*, Sendai, 1994) Berlin, pp. 1–15.

Abramsky, S., Jagadeesan, R., and Malacaria, P.: 2000, 'Full Abstraction for PCF, *Information and Computation.* **163**, 409–470.

Abramsky, S. and McCusker, G.: 1997, 'Linearity, Sharing and State: A Fully Abstract Game Semantics for IDEALIZED ALGOL with Active Expressions', in P. O'Hearn and R. Tennent (eds.), *ALGOL-like Languages, Volume 2*, Boston, pp. 297–329.

Abramsky, S. and McCusker, G.: 1999a, 'Game Semantics', in H. Schwichtenberg and U. Berger (eds.), *Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School*, Berlin, pp. 1–55.

Abramsky, S. and McCusker, G.: 1999b, 'Full Abstraction for Idealized Algol with Passive Expressions', *Theoretical Computer Science* **227**, 3–42.

Abramsky, S. and Melliès, P. A.: 1999, 'Concurrent Games and Full Completeness', in *Proceedings of the Fourteenth International Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, pp. 431–442.

Berry, G. and Curien, P. L.: 1982, 'Sequential Algorithms on Concrete Data Structures', *Theoretical Computer Science* **20**, 265–321,

Blass, A.: 1992, 'A game Semantics for Linear Logic', *Annals of Pure and Applied Logic* **56**, 183–220.

Felscher, W.: 1986, 'Dialogues as a Foundation for Intuitionistic Logic', in D. Gabbay and F. Guenther (eds.), *Handbook of Philosophical Logic*, vol. III, D. Reidel Publishing Company, pp. 341–372.

Girard, J.-Y.: 1987, 'Linear Logic', *Theoretical Computer Science* **50**, 1–101.

Girard, J.-Y.: 1989, 'Towards a geometry of Interaction', in John W. Gray and Andre Scedrov (eds.), *Categories in Computer Science and Logic*, Proceedings of the AMS-

IMS-SIAM joint summer research conference held June 14–20, 1987, University of Colorado, Boulder, with support from the National Science Foundation, Providence [Contemporary Mathematics 92], pp. 69–108.

Harmer, R. and McCusker, G.: 1999, 'A Fully Abstract Game Semantics for Finite Nondeterminism', in *Proceedings of the Fourteenth International Symposium on Logic in Computer Science*, Computer Society Press of the IEEE.

Hintikka, J. and Sandu, G.: 1997, 'Game-theoretical Semantics', in J. van Benthem (ed.), *Handbook of Logic and Language*, Elsevier Science.

Hoare, C. A. R.: 1985, *Communicating Sequential Processes*, Prentice-Hall International.

Honda, K., and Yoshida, N.: 1997, 'Game Theoretic Analysis of Call-by-value Computation', in P. Degano, R. Gorrieri and A. Marchietti-Spaccamela (eds.), *Proceedings, 25th International Colloquium on Automata, Languages and Programming: ICALP '97*, Berlin [Lecture Notes in Computer Science 1256], pp. 225–236.

Hyland, J. M. E. and Ong, C. H. L.: 1993, 'Fair Games and Full Completeness for Multiplicative Linear Logic without the Mix-Rule', Unpublished Manuscript.

Hyland, J. M. E. and Ong, C. H. L.: 2000, 'On Full Abstraction for PCF: I, II and III', *Information and Computation* **163**, 285–408.

Joyal, A.: 1977, 'Remarques sur la Théorie des Jeux a Deux Personnes', *Gazette des Sciences Mathématiques du Quebec* **1**(4).

Jürjens, J.: 1999a, 'Towards Reactive Refinement', contributed talk at the Marktoberdorf Summer School.

Jürjens, J.: 1999b, 'A Category of Processes, Specifications and Refinement', talk at the workshop "Categorical Models of Concurrency", Dresden, October 1999.

Lafont, Y. and Streicher, T.: 1991, 'Game Semantics for Linear Logic', in *Proceedings of the Sixth International Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, pp. 43–50.

Laird, J.: 1997, 'Full Abstraction for Functional Languages with Control', in *Proceedings of the Fourteenth International Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, pp. 58–67.

Lamarche, F.: 1994, 'Sequentiality, Games and Linear Logic (Announcement)', in *Workshop on Categorical Logic in Computer Science*.

Lorenzen, P.: 1960, 'Logik und Agon', in *Atti del Congresso Internazionale di Filosofia*, Sansoni, Firenze, pp. 187–194.

Malacaria, P. and Hankin, C.: 1999, 'Non-deterministic Games and Program Analysis: An Application to Security', in *Proceedings of the Fourteenth International Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, pp. 443–452.

McCusker, G.: 1998, 'Games and Full Abstraction for a Functional Metalanguage with Recursive Types', Berlin [Distinguished Dissertations in Computer Science].

Matthes, R.: 2002, 'Tarski's Fixed-point Theorem and Higher-order Term Rewrite Systems', this volume.

Merz, S.: 2002, 'Model Checking and Beyond: On the Analysis of Reactive Systems', this volume.

Mey, D.: 1994, 'Finite games for a Predicate Logic without Contractions', *Theoretical Computer Science* **123**, 341–349.

Nickau, H.: 1996, 'Hereditarily Sequential Functionals: A Game-Theoretic Approach to Sequentiality', Dissertation, Universität Gesamthochschule Siegen.

Plotkin, G. D.: 1977, 'LCF Considered as a Programming Language', *Theoretical Computer Science* **5**, 223–255.

LFCS
Division of Infomatics,
University of Edinburgh,
U.K.
E-mail: jan@dcs.ed.ac.uk, http://www.jurjens.de/jan